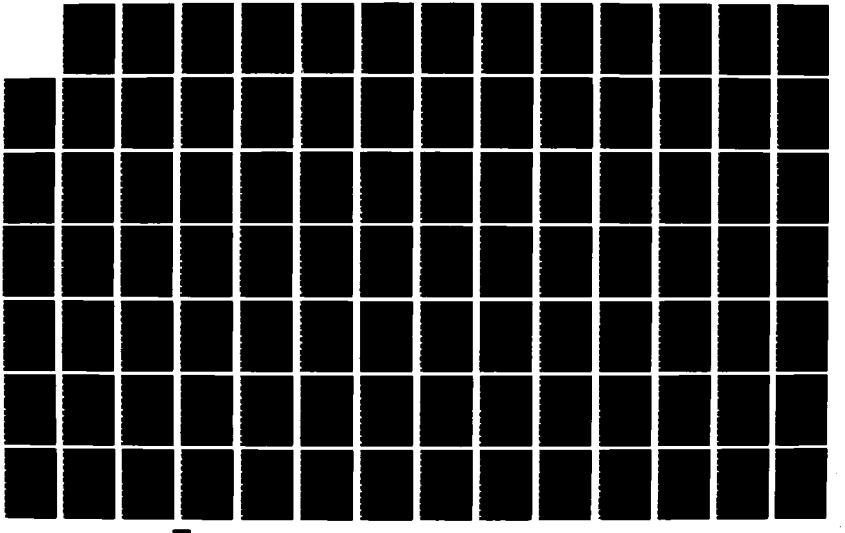
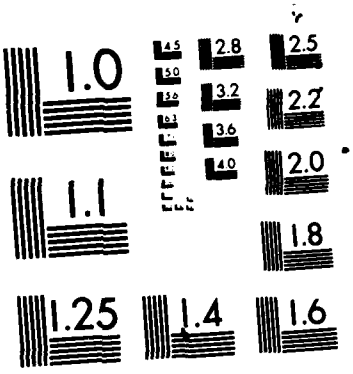
AD-A172 958 FUNCTIONAL PROGRAMMING(U) COMPUTER TECHNOLOGY ASSOCIATES INC LANHAM MD R N MEESON AUG 86 CTA-031-3017001-8602A N00014-84-C-0696 1/2

UNCLASSIFIED F/G 9/2 NL





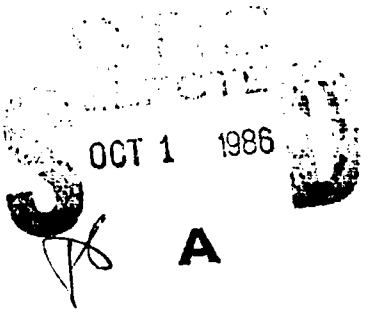
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

AD-A172 958

FUNCTIONAL PROGRAMMING

SBIR FINAL REPORT



DTIC FILE COPY

This document has been approved for public release and sale; its distribution is unlimited

COMPUTER TECHNOLOGY ASSOCIATES, INC.

Denver • Washington, D.C. • Colorado Springs • Albuquerque • San Jose • Ridgecrest • Burlington • Dayton

4/6701 SH

86 9 15 024

12

FUNCTIONAL PROGRAMMING

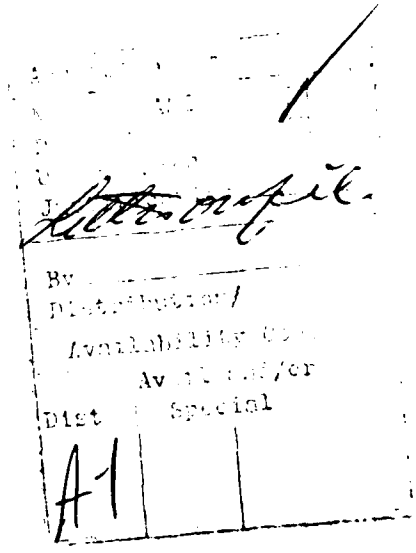
SBIR FINAL REPORT

August 1986

OCT 1 1986

FUNCTIONAL PROGRAMMING
SBIR FINAL REPORT

August 1986



FUNCTIONAL PROGRAMMING

FINAL REPORT

Reginald N. Meeson, Jr.

August 1986

SBIR Phase-II Final Technical Report

prepared for

Office of Naval Research

under

Contract: N00014-84-C-0696
NR SBI-005/6-15-84 (400R)

Computer Technology Associates, Inc.
7501 Forbes Blvd., Suite 201
Lanham, MD 20706
301-464-5300

CTA ~~Ref. No.~~ 031-3017001-8602A

EXECUTIVE SUMMARY

This report describes the accomplishments of an SBIR Phase II project to develop a functional programming language and graphics tools for programming via data flow diagrams. The period covered is from August, 1984 through July, 1986. Our objectives and plans for Phase III are also presented.

Innovations

The chief innovations in this project were the development and full implementation of a functional programming language, and the development of a graphical programming technique with support tools that translate data flow diagrams into executable code.

Accomplishments

The first part of our research was to develop a functional programming language. A compiler and run-time support system for the language described in our Phase II proposal were built and first released in February, 1985. This system was then used experimentally within the company. Several language refinements were made based on user comments, and a second release of the compiler and run-time system was completed in June, 1985. The results of further experimental use and additional enhancements have been incorporated in a final SBIR project release completed in July, 1986.

The second part of our research was to develop a graphical programming technique that would allow programs to be defined via data flow diagrams. We built an interactive, mouse-driven data flow diagram editor and a compiler that generates executable code from the diagrams created using the editor. The first release of the editor was completed in January, 1986. A second release which includes an improved user interface was completed in June,

1986. The data flow diagram compiler was created by extending the functional language compiler and was completed in June, 1986.

In addition to the development of these prototype products, we have written many sample programs to test our system and to demonstrate the practicality of functional programming. Descriptions of the three largest programs we have written are included in this report. These programs illustrate many of the advantages that functional programming provides over conventional programming techniques. Functional programming concepts do "scale up" and can handle large application problems as effectively as simple examples.

Phase III

In Phase III, we plan to produce a commercial version of our language and will focus our initial marketing efforts on the software research and development community. We will continue to work on publishing the results of our research and development efforts, and will conduct seminars and tutorials on functional programming for selected audiences. We will also continue to use our language and graphics tools within the company to specify, design, prototype, and implement software. In addition, we are pursuing research and development support to develop spin-off ideas generated by this project. One of these, for example, is a multiprocessor computer architecture that is based on our run-time software system and is specifically designed for high-speed execution of functional programs.

TABLE OF CONTENTS

1. First Year Accomplishments

Our goals for the first year of this project were to develop a compiler and run-time support system for a functional programming language, and to develop some basic building blocks for our data flow diagramming tools.

Compiler and Run-Time System

Our first efforts were directed at building a compiler and run-time support system for the functional programming language we developed (on paper) in our Phase I project. Our approach to the development of the compiler was to use an automated parser generator (Zuse [1]) and to program translation action routines in Pascal. The compiler produces a low-level form of functional program code, which is then executed by the run-time system. The run-time system was also written in Pascal.

The compiler was constructed by transforming the BNF grammar for our language (see Annex A, Appendix A) into the LL(1) form required by the parser generator. With this grammar and a few pages of Pascal code, we constructed a language recognizer, which performed no translation but could read functional program code and detect syntax errors. The development continued by incrementally adding and testing Pascal code to perform the translation of each component in the grammar. This approach produced a highly reliable and easily modifiable end product.

The run-time system was developed in parallel with the compiler. Its function is to interpret the low-level object code produced by the compiler. The form of this object code (a combinator expression tree) was standardized early in the project, which gave the compiler and run-time system a well-defined interface and has proved quite successful. The run-time system is based on a technique developed by Turner [2]. It executes the

low-level object code by a succession of tree transformations and arithmetic and logical operations.

One of the features of our run-time system is the garbage-collection algorithm. We adopted an algorithm called storage scavenging, which had been developed for SmallTalk [3]. One of the few characteristics our language shares with SmallTalk is that the run-time systems consume storage rapidly, while the programs themselves take up relatively little space. Storage scavenging takes maximum advantage of this and has proved to be very efficient.

An informal evaluation of our functional language and its compiler and run-time system was conducted first by writing a number of functional programs ourselves, and then by distributing the user's guide and software within the company. The comments we got back and our own experience with the language indicated the need for several modifications and enhancements. Chief among these were calls for higher performance, type checking, and better documentation.

The second release of our language contained three features that had been omitted in the original implementation: a name, patterned arguments, and separate compilation. The language was named Ernest -- so that we could say we were programming in Ernest. Patterns provide a very convenient notation for describing structured data such as records, sequences, and trees, which are awkward to manipulate with primitive operations. (See pages 10 and 11 of the Ernest User's Guide in Annex A.) Separate compilation of functions is a natural extension of our original language, since all functions are "pure" code and are independent of any surrounding environment. These extensions were relatively easy to make by modifying the grammar and adding new translation routines.

To improve run-time performance, we added a number of new primitive tree transformation operations and high-level function-forming operations. The new tree transformations are hidden from users but they reduce the size of the intermediate object code and yield faster execution [4]. The most common high-level function-forming operations (construct, filter, generate, map, reduce, and scan) were added to the interpreter as built-in operations. These operations use less storage and run considerably faster than their user-defined counterparts.

In addition to the changes in the software, we completely rewrote the user's guide. The syntax and features of the language were more thoroughly described and the new built-in functions were fully documented.

Data Flow Graphics Groundwork

The first graphics program we developed was to draw data flow diagrams from a detailed specification file. This program could display complete data flow diagrams and provided a basis for the development of the interactive diagram editor. In addition, the specification file was standardized as the interface between the editor and the data flow diagram compiler.

The graphics software was built in several layers. The first layer provided a GKS-like interface to the commercial graphics package we used. The second layer positioned and drew data flow diagram symbols. The third layer added text for symbol labels. The top layer, at this level of development, read an interface file and displayed the specified diagram.

2. Second Year Accomplishments

Our goals for the second year of this project were to improve the capabilities of the Ernest compiler and run-time system, to complete our data flow diagram editor and compiler, and to conduct evaluations of the practicality of functional programming for application software development.

Compiler and Language Extensions

The most significant changes we made in the Ernest compiler were to split it into separate passes. This simplified the programs for each pass and allowed us to experiment with new features more easily. The first pass reads Ernest source code and produces an abstract syntax tree (AST), which is a half-digested form of the original program code. The second pass reads the abstract syntax tree and generates object code for execution. The configuration of these components is shown in Figure 1.

We made significant improvements in pass-one in the reporting of program syntax errors. We also added procedures that attempt to repair simple errors such as missing parentheses, and to resynchronize with program source code after more serious flaws such as omitting the "else" part of a conditional expression. (Earlier versions of the compiler simply stopped at the first error.) In pass-two we added several peep-hole optimizations and found a way to share object code that had been duplicated earlier.

An additional, intermediate stage was to have been added to the compiler to provide type checking. Our original approach was to infer or derive all the necessary type information directly from function definitions and program code, as described by Milner [5]. This approach simplifies function definitions by

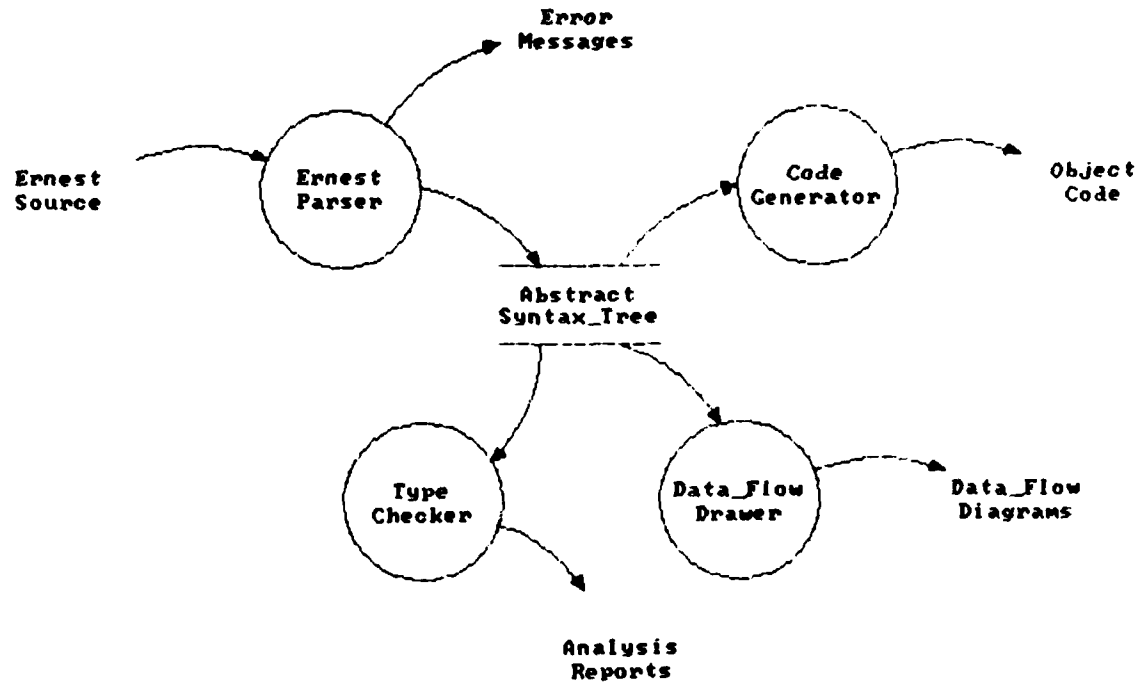


Figure 1. Two-Pass Ernest Compiler Configuration

eliminating redundant type specifications and, at the same time, allows the broadest interpretation of each function's type. The algorithms for this analysis have been developed and tested (see Annex C), but they do not provide a sufficient type mechanism for a practical programming language. Hence, we have not yet incorporated them into the compiler.

The application programs we developed immediately indicated needs for enumerated and variant-record types. Declarations for such types cannot be inferred from function definitions or program code. Ernest must, therefore, be extended to capture this essential information. Enumerated types are straightforward. Records are more challenging because we would like to allow programmers to specify types with the same level of generality (i.e., full polymorphism) as types derived from function defini-

tions. Integrating type declaration facilities into a language, though, must be done carefully and cautiously to avoid the pitfalls of complicating program specifications and unnecessarily restricting type definitions. Furthermore, it may not be possible to make only a "minor" extension for type declarations.

Run-Time System Enhancements

Our highest priority for the run-time system was to improve its performance. Both storage space and execution speed limit the scale of application programs that can be implemented using our system. Hence, we were motivated to make improvements wherever possible. For example, to conserve storage space, we developed a more compact internal representation of sequences and character strings. We also made substantial improvements in both the compiler and the run-time system by streamlining I/O operations.

Before making these changes, we investigated three promising alternative approaches to run-time environments that had been reported in the literature. We found that one, the G-machine [6], would not provide significantly higher performance. The second, supercombinators [7], would require more extensive modifications to the compiler than we were prepared to undertake. The third, threaded code [8], would significantly reduce the portability of our prototype system. Hence, we chose to continue to refine our original approach.

Our second priority for the run-time system was to solve a shortcoming of demand-driven or "lazy" evaluation that prohibits the concurrent generation of multiple output streams, and can cause extensive buffering of input streams. This is not a serious flaw for experimental use of our language, but it would be a severe restriction in a full-scale commercial product.

The problem is that functional languages do not directly control the sequence of input operations or the sequence of expression evaluations. The compiler and run-time system handle these "details." The standard technique, which is called demand-driven or lazy evaluation, is to read input and evaluate results based on demand for output. This concentrates all processing on generating the next output value. In programs that share a data steam between two or more functions, however, this technique insists on evaluating one function completely before it starts on the next. In the meantime, while the other functions that share the stream are suspended, the entire stream must be stored in memory. This can require more memory than is available, which could cause the program to fail without generating any output at all.

Our solution to this problem is a variation of lazy evaluation, which we call "just-in-time evaluation." The concept behind this technique is to evaluate any pair of functions that share a stream in parallel, so that the stream is consumed sequentially and does not have to be stored. In a multiprocessor environment, two such functions could be scheduled so that they consume their common input at the same rate. The effect is similar to the just-in-time scheduling techniques used in industry to facilitate work flow and minimize inventories (i.e., storage). We have completed the detailed design of a scheme that simulates the parallel evaluation of stream-sharing functions in our uniprocessor environment. Its incorporation into the Ernest run-time system, however, has been deferred to Phase III.

Data Flow Diagram Editor

The first major piece of new work completed in the second half of the project was the development of a data flow diagram editor called Van (deGraph Generator). The concept of operations for Van is to allow programmers to specify functions by creating and manipulating data flow diagrams interactively on software

development workstations. The ultimate purpose of the editor is to create a data flow programming environment in which diagrams can be compiled and run like ordinary programs.

Van appears to directly manipulate the diagram displayed on the user's screen. In fact, however, it manipulates a set of internal tables similar to those used in the simple drawing program described earlier. The image displayed on the screen is a projection of the data in these tables. The editing operations supported include:

(1) Adding, deleting, and moving diagram components using the mouse;

(2) Adding, deleting, and modifying diagram nomenclature such as component identifiers and comments; and

(3) Retrieving diagrams to be displayed, edited, or printed, and saving them after they have been created or modified.

Complete descriptions of the editing commands are given in the Van User's Guide in Annex B.

Data flow diagrams capture almost all the information necessary to compile them into executable code. An example of an item that is not represented, though, is the order of function arguments. There is no concept in data flow diagrams that flows enter a process in any particular order. There is no way, for instance, to distinguish between the data flow diagrams for the operations "a divided by b" and "b divided by a." Of course, this is essential information for converting data flow diagrams into executable code. An additional service provided by the editor, therefore, is to collect and record this information and allow the user to change it if necessary.

Van was designed to handle only small diagrams that can be drawn on a single screen. Realistic programs are much larger than this and require multiple levels of definitions. Van, therefore, creates tree structures of single-screen diagrams and

provides "zoom-in" and "zoom-out" operations, which allow users to probe down into process definitions or to take a step back and observe the program from a broader perspective.

With Van, we followed our standard method of building a rough version of the program and then distributing it within the company for evaluation. In this case we had several members of our human factors staff critique Van's user interface. They identified a number of problems (which were not obvious until they showed them to us) and suggested several improvements. All of their suggestions were accepted and have been incorporated in the final release of Van.

Data Flow Diagram Compiler

The second major development in this part of the project was a compiler that translates data flow diagrams into executable code. Current software development techniques that use data flow diagrams for design require a manual conversion of the non-procedural diagrams into procedural programs. The disparity between the procedural and non-procedural views of the problem, however, makes this conversion a difficult, time-consuming, and error-prone task. Our solution is to interpret the diagrams as functional expressions and eliminate the conversion to procedural code.

The front end of our compiler takes the tabular descriptions of data flow diagrams created by the editor and constructs equivalent abstract syntax trees. We then use the second pass of the Ernest compiler to generate executable object code. The configuration of the data flow diagram compiler is shown in Figure 2. This arrangement will also accommodate the intermediate type-checking pass when it is completed.

The construction of abstract syntax trees is relatively straightforward if the data flow diagrams represent pure func-

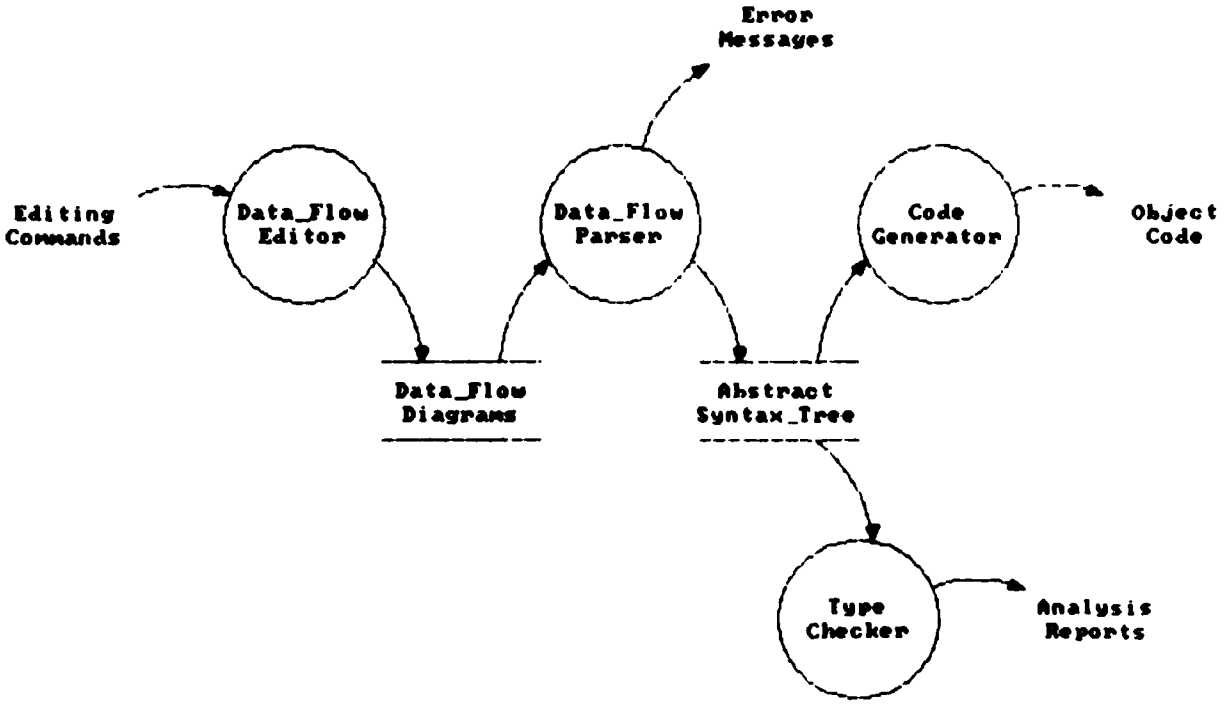


Figure 2. Data Flow Diagram Compiler Configuration

tions. The data flow tables contain all the required structural information in the form of a connectivity matrix. The interface and process tables contain the names of all the parameters and functions referenced. The compiler picks out the formal parameters for the function being defined and builds a parse tree by tracing the structure of the expression represented by the connectivity data.

The only serious difficulties we have found are in compiling data flow diagrams that contain data storage components. Data stores are the only history-dependent components in an otherwise completely side-effect-free technique for specifying computations. They introduce the possibility of unpredictable and non-reproducible program behavior, because there is no way to control the order of updates or references to the stored data. Since

functional programs are free of side effects, their behavior is always reproducible (even if incorrect). Hence, we have had to restrict the use of data stores to a well-controlled set of function-forming operations, namely: "generate", "reduce", and "scan."

Application Programs

To evaluate the utility of functional programming in practical applications, we developed three programs that are significantly larger and more complex than the simple examples we used earlier to test and demonstrate Ernest. Our source for these programs was from company software development projects (including our own), where we have used Ernest for design and rapid prototyping. In this section we briefly summarize each of these projects. More complete reports are included in Annex C.

The first (and largest) of these projects was to develop an interactive tool to analyze the effects of automating job activities on personnel and operations at NASA ground control centers. This analysis is valuable because automating job activities does not always have a positive effect on personnel and overall system performance. The program we developed allows a user to investigate a hierarchical model of the characteristics of functions and tasks associated with control center operations. The user can display data at any level in the model and can experiment with changes in work activities to determine the expected effects on personnel and system performance. This program consists of approximately 1000 lines of Ernest source code for function definitions and approximately 700 lines for the definition of the model data structure.

The second project was to prototype the type checking algorithms that we will use in extending the Ernest compiler. Ernest does not require the programmer to specify any type information in function definitions. The program we developed derives

the most general interpretation of a function's domain and range directly from its defining expression. A key part of the type checker is a unification algorithm similar to those used in interpreters for logic programming languages. The type checker often produces more general and more useful functions than the programmer would have specified. The prototype type checker consists of approximately 400 lines of Ernest source code.

The third project, which is not yet complete, is developing an expert system to support space science research. This work is part of a NASA Phase I SBIR project to evaluate the applicability of expert system techniques in space research. The knowledge to be represented in this system includes the relationship of features in remote-sensed data to the solution of specific scientific problems. This knowledge will be applied to selecting archived data and screening new data for scientific investigations. The inference engine for this expert system consists of approximately 350 lines of Ernest souce code. The scientific knowledge base and inference rules have not yet been developed.

We have recently started a fourth project to implement a collection of signal processing algorithms in Ernest. These algorithms will include Kalman filtering, fast Fourier transforms, and non-linear and adaptive filtering.

3. PHASE-III STRATEGY

Our strategy for Phase III has two major thrusts. The first is to develop and market a line of commercial functional programming products based on the prototypes we have developed in this project. The second is to attract further research and development support to investigate spin-off ideas that have come out of our Phase-I and Phase-II work.

Commercial Product Plans

Our first product goal is to market a functional programming "discovery" kit which will include production versions of our Ernest compiler and run-time system. Our target market segment comprises computer scientists and software engineers who need to maintain (i.e., broaden) their professional skills. The programming languages we have to compete with in this market include, primarily, Prolog and SmallTalk. However, Ernest will be a unique entry because there are no other functional languages on the market.

Each discovery kit will include a tutorial on functional programming, an Ernest User's Guide, and a diskette containing the compiler, run-time system, and example programs. The first offering will be for IBM-PC and compatible microcomputer systems. We will handle initial distribution ourselves until sales can sustain additional distribution channels.

Second-generation products will include higher-performance versions of the Ernest compiler and run-time system and versions for other computer systems such as DEC VAX's, Sun workstations, and Apple Macintoshes. We will also introduce production versions of our data flow programming tools.

Ernest and Van will be promoted through new product announcements, news releases, and limitted but well-focused advertising. We have collected readership profiles and advertising information from the leading professional journals and popular microcomputer magazines. We will register product announcements with all of these publications and will advertise in those that reach the largest number of prospective customers. We also plan to purchase mailing lists from these publishers for direct mail advertising.

Further Research and Development

Research projects frequently raise more questions than they answer. In our case we found a number of interesting ideas that we think are worth further investigation. For example, we observed in Phase I that Ada's *generic* facilities could be used to define function-forming operations [9]. (These operations are an essential component of functional programming languages.) We also observed that Ada's overloading mechanism provides another form of function generalization found in functional languages. We would like to investigate these characteristics further to determine how Ada might be extended to support functional program specifications. Another approach is to see if we can marry Ada and Ernest and gain the advantages of the special features of both languages.

Multiprocessor computer architecture is another area that we would like to investigate further. Machines with literally thousands of processors have already been built using conventional hardware technology. No conventional programming languages, however, are able to support software development for these machines. On the other hand, functional languages offer ideal characteristics for highly parallel execution. They provide no mechanism for specifying sequential operations and they are free of side effects. This allows subexpressions to be evaluated freely in parallel.

Previous approaches to multiporcessor architecture have been almost entirely hardware driven, with little or no thought given to the software required to solve application problems. The result has been a succession of incredibly powerful machines that nobody can program.

We are addressing the multiprocessor design problem with an entirely different approach; namely, by developing the software first. Ernest's run-time system emulates an abstract non-von Neumann machine that executes the object code produced by the compiler. Our multiprocessor architecture concept is to build a machine that can execute our object code directly.

REFERENCES

1. Pyster, A., ZUSE User's Manual, Dept. of Computer Science, Univ. of California, Santa Barbara, TRCS81-04, May 1981.

2. Turner, D., "A New Implementation Technique for Applicative Languages", Software--Practice and Experience, Vol. 9, No. 1, 1979, pp. 31-49.

3. Ungar D., "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm", Proc. ACM Symp. on Practical Software Devel. Environments, April 1984, pp. 157-167.

4. Turner, D., "Another Algorithm for Bracket Abstraction", J. of Symbolic Logic, Vol. 44, No. 2, June 1979, pp. 267-270.

5. Milner, R., "A Theory of Type Polymorphism in Programming", J. of Computer and System Sciences, Vol. 17, 1978, pp. 348-375.

6. Johnsson, T., "Efficient Compilation of Lazy Evaluation", Proc. SIGPLAN '84 Symp. on Compiler Construction, June 1984, pp. 58-69.

7. Hughes, J., "Super-Combinators, A New Implementation Method for Applicative Languages", Conf. Record 1982 ACM Symp. on LISP and Functional Programming, August 1982, pp. 1-10.

8. Loeliger, R., Threaded Interpretive Languages, Byte Books, Peterborough, New Hampshire, 1981.

9. Meeson, R., "Function-Level Programming in Ada", Proc. IEEE CS 1984 Conf. on Ada Applications and Environments, October 1984, pp. 128-132.

Annex A. ERNEST USER'S GUIDE

ERNEST

Functional Programming

User's Guide

Version *iii*

July 1986

Ernest

Functional Programming

User's Guide

Version iii

July 1986

Reginald N. Meeson, Jr.

Computer Technology Associates, Inc.
7501 Forbes Blvd., Suite 201
Lanham, MD 20706
301-464-5300

This research was sponsored by the US Navy, Office of Naval Research under contract number N00014-84-C-0696.

Copyright (c) 1986, Computer Technology Associates, Inc.

Permission to copy this document for personal or educational use is granted provided that copies are not made or distributed for commercial advantage.

TABLE OF CONTENTS

PREFACE

This manual is a guide to a functional programming language called Ernest. Ernest is intended to be a general purpose language. That is, we have not tried to focus on any particular application area or guess how people will use it.

This manual is not intended to be a tutorial on functional programming. The example programs presented illustrate many of the basic concepts and we hope they provide sufficient guidance for the diligent reader. A more comprehensive tutorial is in preparation.

Both the language and its implementation will continue to evolve. The primary reason for distributing the present release of this package is to introduce the concepts of functional programming and to determine the product's readiness for commercial release.

We invite your comments. Once you have had a chance to try out our language, we would like to know how you think we could improve it. That includes functional programming in general, Ernest in particular, the compiler and interpreter, and this user's guide. We will attempt to incorporate all viable suggestions as we continue to develop this product.

Development Status

The compiler and run-time interpreter were developed as part of the prototype implementation phase of a Small Business Innovative Research (SBIR) project sponsored by the US Navy, Office of Naval Research. This software is written in ISO Standard Pascal -- except for the dynamic storage management code -- and runs on IBM-PCs and compatible machines. The programs require at least one disk drive and 128K of memory.

The following features are currently supported:

- Integer Arithmetic ("+", "-", "*", "/", mod)
- Logical Operations ("&", "|", "~")
- Relational Operations ("=", ">", ">=", etc.)
- Sequence Operations (head, tail, ":", "^")
- Boolean, Character, Integer, and String Data
- Function-Forming Operations
- Separate Compilation

Type checking, real arithmetic, graphics primitives, and file I/O are on the list of things to be added.

Distribution Package

The current distribution package consists of this manual and a single 360KB diskette. The diskette contains the following files:

OLIVER1.EXE	GCD.FUN	GCD.HBC
OLIVER1.TBL	RANDOM.FUN	RANDOM.HBC
OLIVER2.EXE	PRIMES.FUN	PRIMES.HBC
OLIVER2.TBL	LENGTH.FUN	LENGTH.HBC
SHERBERT.EXE	REVERSE.FUN	REVERSE.HBC
FUNGO.BAT	WORDCNT.FUN	WORDCNT.HBC
	SORT.FUN	SORT.HBC

The files in the first column contain the compiler, the run-time interpreter, and a convenient batch command file for compiling and executing programs. Their operation is described in Appendix D. The ".FUN" files contain the example functional programs discussed in Section 3. The ".HBC" files contain the compiled object code for these programs.

Manual Organization

The material presented in this manual is organized in four major sections. In the first section we present an overview of functional programming, which introduces some of the general concepts and includes some background on the development of the functional approach.

The second section presents the rules for formulating programs in Ernest. This includes discussions of arithmetic and logical expressions; sequences, streams, and strings; applicative function definitions; and function-forming operations.

In the third section we present a series of example functional programs, which include numerical and non-numerical applications.

The fourth section presents a brief set of guidelines which we hope will help new users to develop functional programs.

In addition to these major sections, there are five appendices which include a BNF grammar for Ernest's syntax, a list of predefined library functions, definitions of built-in function-forming operations, directions for compiling and running programs, and a compendium of error messages and their possible causes.

Reg Meeson
Lanham, MD

July 1986

1. OVERVIEW OF FUNCTIONAL PROGRAMMING

Ernest is a programming language that supports a very high level, non-procedural approach to software specification. Ernest provides facilities for defining simple functions, plus a set of powerful function-forming operations for constructing more complex functions and complete programs. Ernest operates entirely with functions and does not support the definition of step-by-step procedures.

This approach encourages definition of reusable components. In fact, program components can easily be parameterized to create general-purpose components that can be tailored to specific applications when needed. Reliability and reusability are further enhanced by the fact that functions have no side-effects and, hence, can cause no hidden changes (i.e., surprises) in the state of a computation.

Background

In his 1977 Turing Award Lecture [1], John Backus introduced some of the radically new ideas of functional programming. He presented two very clear cases against conventional programming techniques. First, he argued that it is extremely difficult to reason about the mathematical and logical properties of conventional programs. This is because conventional languages typically do not follow simple laws of algebraic equivalence or transformation. For example, identical code segments may produce very different results when executed at different times or in different parts of a program. It becomes very difficult, therefore, to build larger programs from smaller components because fundamental principles such as substitution cannot be relied upon.

Backus's second point was that the conventional concept of controlling a machine's actions is completely out of keeping with modern developments in highly parallel non-von Neumann computer architecture. The step-by-step control imposed by conventional programs can easily thwart the use of available parallelism in these advanced machines. At best, extra compile-time analysis is required to determine when sequential commands can be performed in parallel.

In a 1979 paper [2], David Turner described a new implementation technique for functional languages based on theoretical work done in the 1930's by Haskell Curry [3]. Curry developed what he called the Combinatory Logic and showed it to be equivalent to Church's Lambda Calculus. (The Lambda Calculus was developed at about the same time and is well known as a theory of computation and as the basis for LISP.) Turner showed that a modern implementation of Curry's theory had several advantages over existing techniques for interpreting functional programs. A

number of functional languages, including Ernest, have been implemented based on Turner's approach.

General Concepts

The basic components of functional programs are **expressions** and **functions.** This contrasts with conventional languages where the basic components are **statements** and **procedures.**

The use of expressions for arithmetic and logical computations is well established in conventional high-level programming languages. Indeed, the introduction of expressions in high-level languages has been a key factor in increasing software reliability and programmer productivity. It is common practice to let compilers handle low-level details such as register allocation, temporary storage for intermediate results, and generating the actual processing steps necessary to evaluate expressions.

At the next level above expressions, however, traditional high-level languages, including Pascal and Ada, revert to step-by-step statements or commands which operate on individual data items. The same difficulties that hinder assembly language programming make larger Pascal and Ada programs time-consuming to develop and difficult to modify.

Functional programming extends the use of expressions to create and manipulate functions the way conventional languages treat numeric and other data. The principal extension is the addition of **function-forming operations.** Mathematicians call these function-forming operations functionals, which is where we derive the term functional programming. Function-forming operations replace conventional program structures such as assignment statements, conditional statements, loops, and procedure calls.

The correspondence between arithmetic and functional expressions is illustrated in Figure 1, which shows the translation of two expressions into procedural code. The first expression is a simple arithmetic formula which represents a numeric value that can be computed when values for A, B, and C are given. The translation of this expression into assembler code is shown on the right.

The second example is a functional expression which is equivalent to the Pascal code shown on the right. The symbol "o" represents function composition, which is probably the most familiar function-forming operation. "Map" is another function-forming operation which provides a common looping mechanism similar to the Pascal "while" loop in this example. ("Map" is further described in Section 2.)

The relative simplicitiy of the expressions on the left, compared to the more complicated statements on the right can easily be seen. Note that the arithmetic expression can be understood without simulating its evaluation. The same is true

Arithmetic Expression	Assembler Code
A * B + SIN(C)	L 2,A M 2,B LA 1,C BALR 15,SIN A 2,0

Functional Expression	Pascal Code

map(f o g)	read(x);
	while not eof do begin
	write(f(g(x)));
	read(x)
	end

Figure 1. Expressions vs. Procedural Code

for functional expressions once the notation and function-forming operations become familiar. This makes verifying the correctness of functional programs much easier than checking step-by-step procedural code.

The same kind of argument can be made for the relative ease of modifying functional programs. Changes in arithmetic expressions in conventional high-level languages can be made without considering their impact on register allocation or storage for intermediate results. Changes in functional programs can be made without considering their impact on local variable declarations or the restructuring of step-by-step operations. Hence, rather sweeping changes can be made with high confidence when they are required.

2. ERNEST SYNTAX AND FEATURES

This section introduces the syntax and features of Ernest. The terminology and examples in this discussion assume the reader is familiar with conventional high-level languages such as Pascal and Ada. Ernest is presented in a bottom-up fashion, starting with the simplest components and working up to complete programs. A BNF grammar for Ernest is included in Appendix A.

Identifiers, Symbols, and Literals

Identifiers are used to designate constant values, functions, and function parameters. They follow the usual rules of beginning with a letter, which may be followed by additional letters, digits, and underscores. Differences in upper- and lower-case letters are ignored, so that "IEEE" and "ieee", for example, represent the same object. Ernest reserves the following identifiers for expression delimiters, built-in functions, and predefined literal values:

and	false	mod	true
define	if	o	where
else	input	then	undefined

The following special symbols are used in expressions:

"	&	'	(	)	*	+	,
-	--	/	^	/=	:	;	<
<=	=	>	>=	[	]	|	~

The symbol "--" begins a comment, which then extends to the end of the line. The functions of other symbols are described in the sections below.

Conventional notation is used for literal integer and string values. Single quotes (apostrophes) surround strings. Literal values for sequences and tuples are enclosed in angle brackets ("<" and ">") and separated by commas. An example tuple value is:

```
< 'Ernest', <'July',1986>, true >
```

Arithmetic and Logical Expressions

Every attempt has been made to make Ernest's arithmetic and logical expressions as "ordinary" as possible, so that expressions actually mean what they intuitively appear to mean. The precedence of operators, in decreasing order, by category, is shown in Figure 2. As always, parentheses can be used to over-

ride the precedence of operators.

Function applications are also ordinary looking. They consist of a function name followed by arguments in parentheses and separated by commas, as in

```
gcd( 123, 456 )
```

Conditional Expressions

Ernest supports conditional processing in the form of if-then-else expressions similar to those in Algol. (Somehow these useful expressions disappeared in Pascal and Ada.) They have the form:

```
if condition then result_1 else result_2
```

-	Arithmetic negation	(unary prefix)
~	Logical complement (not)	
*	Multiplication	(binary infix)
/	Division	
mod	Modulo (remainder)	
+	Addition	
-	Subtraction	
<	Less than	
<=	Less than or equal	
=	Equal	
/=	Not equal	
>	Greater than	
>=	Greater than or equal	
&	Logical conjunction (and)	
|	Logical disjunction (or)	
:	Sequence construction (cons)	
^	Sequence concatenation	

Figure 2. Precedence of Operators

The "condition" can be any Boolean expression; "result_1" and "result_2" can be arbitrary expressions. The value of "result_1" is returned if the condition is true, and the value of "result_2" is returned if the condition is false. Expressions must always be given for both "then" and "else" results in a conditional expression. An example conditional expression is:

```
if  x >= 0  then  x  else  -x
```

Sequences, Streams, Strings, and Tuples

Sequences, streams, strings, and tuples are all almost the same things. They are all ordered collections of elements. The terms "sequence" and "stream" (and sometimes "list") are used interchangeably to indicate collections of objects of the same type. Strings are sequences of characters, and are so common they have their own syntax for literal values. Tuples are collections of objects of mixed type. Sequences and tuples are usually thought of as finite objects, whereas streams (e.g., I/O streams) are usually thought of as infinite (or at least very long) objects. Ernest makes no distinction -- except that infinite streams can not be sorted or reversed, etc., because these operations require access to all the components of the sequence. If processing an infinite stream requires access to only a finite segment at any time, Ernest can handle it quite easily.

Primitive operations on sequences include "head", "tail", "^", and ":". "Head" is the function that returns the first element of a sequence. "Tail" returns the sequence that remains after removing the head. The infix operator ":" (pronounced "cons") constructs a new sequence with its left-hand argument at the head and its right-hand argument for a tail. The infix operator "^" concatenates two sequences. These operations have the following properties: ("<>" represents the empty sequence)

```
head( <> )  =  undefined    -- (run-time error)
tail( <> )  =  undefined    -- (run-time error)
head( x:t )  =  x
tail( x:t )  =  t
head( s^t )  =  if s=<> then head(t) else head(s)
tail( s^t )  =  if s=<> then tail(t) else tail(s)^t
```

Literal values for sequences and tuples have the following definitions. Note that ":" is right associative, as indicated by the parentheses.

```
< x >  =  x : <>
< x, y, z >  =  x : y : z : <>
             =  x : (y : (z : <>))
```

Function Definitions

There are two forms of function definition in Ernest: conventional definitions which describe a function in terms of operations on a set of formal parameters, and function-level definitions which construct functions using function-forming operations. The basic form of a conventional definition is:

```
function_name ( parameter_list )  =  expression
```

where "function_name" is an identifier, "parameter_list" is a list of parameters separated by commas, and "expression" is an arbitrary Ernest expression. For example, the absolute value function can be defined as

```
abs(x)  =  if x>=0 then x else -x
```

which shows how to test a hypothetical input value (represented by x) and prepare the result.

Functions may be recursive. Recursion, however, is not the only (nor the best) looping mechanism in Ernest. Alternative techniques using function-forming operations are described below.

Subordinate Definitions. Complex functions can often be simplified by introducing local supporting functions to handle lower-level details and by defining local "variables" for common subexpressions. (Variables in functional programs are not storage locations, they are merely identifiers for expressions.) Ernest provides a "where" clause for defining local functions and variables within a function definition. The complete form for function definitions is

```
function_name ( parameter_list )  =  expression
    where  definition_1
    and  definition_2
    ...
    and  definition_n
```

"Where" and "and" clauses are optional. The following example shows the use of a "where" clause to define a local function to help in generating a stream of pseudo-random numbers.

```
random(n) = n : random( next(n) )
    where  next(n) = (31*n + 378) mod 1013
```

This function generates an infinite sequence of numbers that range between 0 and 1012. The parameter "n" serves as the initial seed for the random sequence. Successive values are produced by the function "next" which computes a simple product-remainder formula.

Subordinate functions may be mutually recursive and may invoke the function they help define.

Nested Definitions. "Where" clauses may be nested. This mechanism is required for structuring large programs. However, it introduces two minor complications. The first is that local functions and variables may reuse identifiers that have already been declared. The search for the definition of an identifier that appears in an expression proceeds as follows:

1. Look for it as the name of a subordinate function or variable;
2. If that fails, look for it as a parameter of the current function;
3. Otherwise, it must be defined at a higher level, so move up to the next level and repeat the search;
4. End the search when you have looked through the lists of predefined values and built-in operations given in Appendix C.

One of the consequences of these rules is that subordinate functions may be mutually recursive.

The second complication in nested definitions is due to the ambiguity as to which "where" the "and" clause belongs in the two possible patterns below.

```
...                          ...
where  f(x) = ...            where  f(x) = ...
   where  g(y) = ...            where  g(y) = ...
   and  h(z) = ...           and  h(z) = ...
```

Ernest ignors the indentation and assumes the left-hand interpretation; i.e., that "g" and "h" support the definition of "f". To get the right-hand interpretation, Ernest allows a nested "where" clause to be terminated by a semicolon, as shown below.

```
...
where  f(x) = ...
   where  g(y) = ... ;    -- ends definition of "f"
and  h(z) = ...
```

Patterns. When functions take sequences or tuples as arguments, it is often desirable to name the components of these structures so they can be used in the defining expressions. The

following patterns for structured parameters are recognized:

```
parm_1 : parm_2          -- for non-empty sequences, and
< parameter_list >       -- for fixed-length tuples
```

"Parm_1" and "parm_2" are identifiers or patterns and "parm_list" is a list of identifiers or patterns separated by commas. An example use of this is in the definition of a function to compute the magnitude of vectors represented by <x,y> coordinate pairs.

```
magnitude( <x,y> )  =  sqrt( x*x + y*y )
```

Patterns can also be used in "where" clauses to define structured variables. A common use of this is to define functions on sequences that first test for the empty sequence and then, on non-empty sequences, operate on the head and tail components. An example of this is:

```
sum(s) = if s=<> then 0 else x + sum(t)
   where  x:t = s
```

Here, "x" and "t" serve as convenient names for "head(s)" and "tail(s)", respectively. Note also that "x" and "t" are undefined if the sequence "s" is empty.

Function-Forming Operations

Function-forming operations are functions or expressions that yield functions as results. Probably the most familiar example of this type of operation is function composition, which is a primitive operation in Ernest. The symbol "o" can be used to compose functions as in the following definition:

```
safe_sqrt  =  sqrt o abs
```

This definition is equivalent to

```
safe_sqrt(x)  =  sqrt( abs(x) )
```

In fact, identical object code is generated for both definitions. The difference between them is significant, however. The first definition is a very simple example of a much more powerful form of programming which directly manipulates the component functions. **This is functional programming!** The second definition is weaker in the sense that it must introduce an extraneous "dummy" parameter and describe the function in terms of operations on data.

One of the big advantages of function-forming operations and function-level expressions is that complex functions can often be

defined more easily by such operations than by conventional definitions. For instance, many functions which require recursive conventional definitions can be constructed by functional expressions without explicit use of recursion. Several examples of such functions are presented below and in Section 3.

Partial Application. Any function with more than one argument can be considered a function-forming operation in Ernest. The reason for this is that functions are evaluated by applying them to one argument at a time, from left to right. The result of consuming the first (left-most) argument is a new function which is then applied to the next argument and so on, until all the arguments are consumed.

The most common use of this language characteristic is to "partially" apply a function to its first few arguments, leaving later arguments uncommitted. A simple example of this is the successor function "succ" which adds 1 to its argument:

```
succ  =  "+"( 1 )
```

This function fixes one of the arguments for the addition operation by partial application and leaves the second argument to be filled in later. An equivalent conventional definition for this function is

```
succ( x )  =  x + 1
```

Built-In Function-Forming Operations. The key to the utility of partial application is the wide variety of functions that can be constructed using a small number of function-forming operations. Several of these operations that have found their way into common use have been included as built-in functions in Ernest. These functions are briefly described in Figure 3. The figure shows several examples of iterative operations that can be described without explicit use of recursion. Actually, the recursion is hidden within the definitions of these function-forming operations. For example, the definition of "map" in Ernest is

```
map( f, s )  =  if s=<> then <>
                else f(x):map(f,t)
   where  x:t = s
```

Further examples of the use of these operations are presented in Section 3.

Generate: applies a function repeatedly to produce an infinite sequence.

```
generate( f, x )  =  < x,  f(x),  f(f(x)),  ... >
```

An example where "generate" could be used is in

```
random  =  generate( next )
where  next(m) = (31*m + 378) mod 1013
```

"Random" is a function which, when applied to an integer argument, produces an infinite sequence of pseudo-random numbers. "Random" passes on its argument as the second argument to "generate"; i.e.,

```
random( n )  =  generate( next, n )
```

Map: applies a function to each element of a sequence, producing a new sequence.

```
map( f, <a,b,c,...> )  =  <f(a),f(b),f(c),...>
```

"Mapping" is a common operation on streams. For example, a function that converts all the upper-case letters in a character stream to lower case is

```
lowercase  =  map( cvt )

where  cvt(ch) = if ch<'A' | ch>'Z' then ch
                 else chr(ord(ch)+32)
```

Filter: selects the elements of a sequence that satisfy a given predicate. For example,

```
filter( odd, <3,8,5,2,7> )
where  odd(n)  =  n mod 2 = 1
```

produces the sequence <3,5,7>.

Figure 3. Ernest Function-Forming Operations

Reduce: accumulates the result of applying a binary function "between" sequence elements. For example,

reduce("+", 0, <a,b,c,...>) = 0+a+b+c+...

This is particularly useful as a function-forming operation. A simple example is the following function definition:

sum = reduce("+", 0)

Construct: applies a sequence of functions to a common argument, producing a tuple of results. An example of this is:

construct(<f,g,h,...>,x) = <f(x),g(x),h(x),...>

This is such a useful function-forming operation it has been given its own special syntax. Square brackets around a list of functions have the following meaning:

[f, g, h, ...] = construct(<f,g,h,...>)

Scan: applies a binary function between sequence elements similar to "reduce", but produces the sequence of all the partial results. For example,

scan("+", 0, <a,b,c,...>) =
< 0, 0+a, 0+a+b, 0+a+b+c, ... >

This function-forming operation is a very general one. In fact, both "map" and "reduce" can be defined in terms of it. For example, the last element produced by "scan" is the value produced by "reduce." Hence,

reduce(f,a) = last o scan(f,a)

where last(x:t) = if t=<> then x
else last(t)

"Last" returns the last element in a non-empty sequence; "o" is the infix operator for function composition.

Figure 3. Ernest Function-Forming Operations (cont.)

Input and Output

The standard input character stream (e.g., from the user's keyboard) is available through the predefined stream named "input." All output is directed to the standard output stream.

Program Structure

Complete programs are formed in Ernest by a series of (zero or more) function definitions, followed by an expression. Functions defined at the program level are global in scope and may be compiled separately. The reserved word "define" is used to introduce each global function definition. The order of global function definitions in a program is not significant. The form of a complete program, therefore, is

```
define  definition_1
...
define  definition_m
expression
```

3. EXAMPLE FUNCTIONAL PROGRAMS

This section presents a number of simple functional programs which can be run on our system and are included on the distribution diskette. The examples range from simple numerical computations to string manipulation and sorting. A complete, executable specification is given for each program.

At the beginning of each program we define general-purpose utility functions and function-forming operations. Special-purpose functions which are unique to the problem are specified in local "where" clauses. This style of programming is intended to promote the development of reusable global function definitions.

Numerical Programs

The following examples illustrate simple numerical computations. They include computing the greatest common divisor of two numbers, generating random and prime numbers, and computing the dot product of two vectors.

Greatest Common Divisor. The greatest common divisor of two numbers is the largest value that divides both numbers evenly. Euclid's algorithm for finding the "gcd" of two positive integers is used in the first version of this program:

```
-- Compute the greatest common divisor

gcd( 105, 45 )

where  gcd(a,b) = if a=b then a
                  else if a>b then gcd(a-b,b)
                  else gcd(a,b-a)
```

Pseudo-Random Numbers. This program is virtually identical to the example application of "generate" discussed on page 12. "List" inserts commas and spaces between sequence elements for readability.

```
define  list(s) = '<' : if s=<> then <'>'>
                        else head(s) : lst(tail(s))

     where  lst(s) = if s=<> then <'>'>
                     else ',' : ' ' : head(s)
                              : lst(tail(s))
```

```
-- Generate an infinite sequence of random numbers

list( random(123) )

where  random = generate( next )

     where  next(n) = (31*n + 378) mod 1013
```

Prime Numbers. The objective of this program is to generate the sequence of all the prime numbers. Several functional solutions to this problem have been described in the literature (cf., [4] and [5]). The one presented here produces the longest sequence before running out of storage space.

```
define  list(s) = '<' : if s=<> then <'>'>
                        else head(s) : lst(tail(s))

     where  lst(s) = if s=<> then <'>'>
                     else ',' : ' ' : head(s)
                          : lst(tail(s))


-- Generate the sequence of all the prime numbers

list( primes )

where  primes = 2 : genprimes(3)

   where  genprimes(n) = if divisible(n)
                            then genprimes(n+1)
                         else n : genprimes(n+2)

      where  divisible(n) = dvbl(n,primes)

         where  dvbl(n,x:t) = if n mod x = 0 then true
                              else if x*x > n then false
                              else dvbl(n,t)
```

"Genprimes" generates the sequence of prime numbers starting with the first prime that is greater than or equal to its argument. "Divisible" tests its argument to see if it is divisible by any previously generated prime.

Vector Dot Product. The dot product of two vectors is defined as the sum of the pairwise products of vector elements.

```
define  pairwise(f,a,b) = if a=<> & b=<> then <>
                          else f(al,bl) :
                               pairwise(f,an,bn)

     where  al:an = a     and  bl:bn = b
```

```
define  sum = reduce("+",0)

-- Compute the dot product of two vectors

dotprod( <1,2,3>, <4,5,6> )

where  dotprod(a,b) = sum( pairwise("*",a,b) )
```

"Pairwise" applies a function to the elements of two input sequences, taking one value from each sequence at a time and producing a new sequence as a result. That is,

```
pairwise( f, <a1,a2,...>, <b1,b2,...> )  =
                < f(a1,b1), f(a2,b2), ... >
```

The definition of "sum" in terms of "reduce" was discussed on page 13.

The heart of this program is the expression

```
sum( pairwise("*",a,b) )
```

which reads almost exactly like the English definition of dot product. This correspondence between definitions is another reason why function-level programming is easier, faster, and more reliable than conventional techniques.

Non-Numerical Programs

The following examples deal primarily with functions that transform or rearrange sequences. They include finding the length of a sequence, reversing and appending elements to sequences, counting the number of words in text, and sorting.

Length. One of the simplest operations on sequences is to find their length. A non-recursive definition can be formulated using the function-forming operation "reduce":

```
define  length = reduce(count,0)

     where  count(n,x) = n+1

-- Find the length of a sequence

length( 'How now brown cow?' )
```

"Count" simply increments a counter for each element in the input sequence.

Reverse and Append. The following functions reverse the elements in a sequence and append a new element at the end of a sequence.

```
define  list(s) = '<' : if s=<> then <'>'>
                        else head(s) : lst(tail(s))

     where  lst(s) = if s=<> then <'>'>
                     else ',' : ' ' : head(s)
                              : lst(tail(s))

define  reverse = reduce(snoc,<>)

     where  snoc(s,x) = x:s

define  append(a,s) = s ^ <a>


--  Example sequence transformations (contrived)

list( < reverse('elloH'),
        'how are ' ^ append('?','you') > )
```

"Reverse" is defined using "reduce." The function "snoc" is the same as "cons" with its arguments reversed. "Append" is defined by forming a sequence containing the new element and concatenating it to the end of the original sequence.

Word Count. The purpose of this program is to count the number of words (i.e., contiguous sequence of non-blank characters) in a text string.

To simplify this problem, we have split it into two basic steps. First, the input string is transformed into a string containing asterisks, dashes, and blanks. Asterisks replace the first letters of words, dashes replace subsequent letters, and blanks remain blanks. That is, the input is first transformed into the string:

```
'*-- *-- *---- *---'
('How now brown cow?')
```

The second step is simply to count the asterisks.

```
define  length = reduce(count,0)

   where  count(n,x) = n+1
```

```
-- Count the number of words in a text string

wordcount( 'How now brown cow?' )

where  wordcount = length o filter(equal('*'))
                          o scan(markword,' ')

     where  markword(a,b) = if b=' ' then ' '
                            else if a=' ' then '*'
                            else '-'
```

"Wordcount" is defined by the composition of three functions which form a pipeline of sequence transformation. The first transformation is the tricky one. All it requires, though, is the observation that the beginning of a word can be distinguished by looking between successive characters in a string. A new word starts when a blank is followed by a non-blank. This is a perfect scenario for "scan"; and once it is recognized, the problem is reduced to defining the function "markword."

The second transformation is to screen out all the asterisks so they can be counted. "Equal('*')" is a function which tests a single character to see if it is an asterisk. Filtering with this function produces a subsequence of the input containing all the asterisks. The length of this subsequence, therefore, equals the number of words in the original text string.

Sorting. No description of a programming language is complete without a discussion of sorting. This program illustrates the technique of inserting elements, one at a time, into a new sequence in the desired order.

```
define  list(s) = '<' : if s=<> then <'>'>
                  else head(s) : lst(tail(s))

     where  lst(s) = if s=<> then <'>'>
                     else ',' : ' ' : head(s)
                              : lst(tail(s))

define  sort(order) = reduce( insert(order), <> )

     where  insert(p,s,x) = if s=<> then <x>
                            else if p(x,y) then x:s
                            else head(s):insert(p,t,x)

          where  y:t = s


-- Sort a sequence of numbers in ascending order

list( sort(ascending,<3,-8,5,-2,7>) )
```

```
where  ascending(x,y) = x <= y

   --  descending(x,y) = x >= y
```

"Insert" searches an already sorted sequence and inserts a new element in its proper position. Ascending and descending sorts can be accomplished by supplying the function used to decide where an element belongs in the sorted sequence.

"Reduce" is used to orchestrate the insertion of each element in the input sequence into a growing partial sequence of sorted elements. When the final element has been inserted, "reduce" returns the complete sorted sequence.

4. PROGRAMMING IN ERNEST

In this section we offer some general guidelines which we have found useful in developing functional programs. While we feel these guidelines are useful, we also think they can be expanded and improved. Hence, we are not yet ready to promote them as a complete methodology.

Top-Down Design

A top-down design approach is essential to breaking problems into manageable pieces. Separate the problem's functional, operational, and performance requirements. One of the problems of conventional programming techniques is that function, performance, and operations issues are typically all jumbled together. Functional programming requires focusing on the program's **function.**

Identify the necessary program inputs and the required program outputs, then characterize the program as a function from inputs to outputs. If several outputs are to be produced, consider each one to be the result of a separate function.

Functional programming offers more flexibility in partitioning problems than conventional programming languages. One example of this is illustrated in our solution of the word-count problem on page 18. This program first transforms its entire input stream, then filters out selected elements, and finally counts what is left. Conventional programming techniques would probably not have led to this solution. However, it is easy to understand, it is easily verified, and it is just as efficient as more conventional solutions.

Bottom-Up Implementation

We recommend a constructive, bottom-up approach to writing programs. Simple functions can be quickly built and tested before being incorporated into larger ones. Functions at any level can be easily combined to form more complex functions. They can also be taken apart just as easily to correct errors and to form new functions. Conventional programming languages do not have this flexibility because of the changes in program variable declarations and initialization that would be required to match the changes in functional organization.

Data Flow Diagrams

We have found data flow diagrams [6] to be extremely helpful in developing functional programs. They provide an intuitive graphical representation of the construction of functions. The

translation between data flow diagrams and program code is relatively simple. This allows designs to be easily converted into executable programs. Similarly, programs can be easily converted into data flow diagrams for verification and documentation. We hope to provide tools to facilitate data flow programming with the next release of our compiler.

Recursive Functions

While Ernest fully supports recursive functions, most programmers (except for those with an affinity for quiche) have difficulty with recursion. Our recommendation is to avoid recursion where ever possible and use function-forming operations instead. Ernest's built-in function-forming operations cover a wide range of applications. For example, "reduce" can be used to sort numbers as well as sum them. As an added incentive, functions constructed using the built-in operators usually run faster than equivalent recursive functions.

Function-Forming Operations

Try to match Ernest's function-forming operations to the structure of the problem. This may take some practice but it is the key to building functional programs. If a function-forming operation can be used, then proceed to develop the lower-level functions required to construct the solution.

We have found that developing new function-forming operations with the desired level of generality requires considerable analysis and, hence, must be accorded more respect than developing ordinary functions. This is because an effective function-forming operation must abstract a general functional structure from the details of specific applications. An example of a well-designed function-forming operation is "map." This operation can be used in many different applications because it is cleanly separated from the details of any individual program. Functional programming provides powerful facilities for abstraction and, when used correctly, yields simple solutions to complex problems.

REFERENCES

1. Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", Comm. ACM, Vol. 21, No. 8, August 1979, pp. 613-641.

2. Turner, D., "A New Implementation Technique for Applicative Languages", Software--Practice and Experience, Vol. 9, No. 1, 1979, pp. 31-49.

3. Curry, H.B., and R. Feys, Combinatory Logic, North-Holland, Amsterdam, 1958.

4. Henderson, P., Functional Programming Application and Implementation, Prentice-Hall International, 1980.

5. Darlington, J., P. Henderson, and D. Turner (eds.), Functional Programming and its Applications, Cambridge Univ. Press, 1982.

6. DeMarco, T., Structured Analysis and System Specification, Yourdon, Inc., New York, 1979.

Appendix A. ERNEST BNF GRAMMAR

In this appendix we present a BNF grammar for Ernest. Some of the particular details of this meta-notation are:

- Literal symbols and reserved words are shown in quotes, such as 'define', '(', ')', etc.

- Square brackets enclose optional items; for example,

  [Where_Clause]

  indicates that "where" clauses are optional.

- Braces enclose a repeated item, which may appear zero or more times; for example,

  { ',' Parameter }

  indicates that a parameter list may have an arbitrary number of parameters separated by commas.

- Vertical bars separate alternative items, as in

  Identifier | Structure

  which indicates that parameters may be either simple identifiers or structures.

- The terms "Character", "Identifier", "Number", and "String" represent primitive lexical groups which are not defined in this grammar.

The axiom for the grammar productions is the non-terminal "Program".

```
Program      ::=  { 'define' Definition }
                  Expression  [ Where_Clause ]

Definition   ::=  Identifier [ '(' Parameter_List ')' ]
                      '=' Expression [ Where_Clause ]

               |  Structure '=' Expression

Parameter_List  ::=  Parameter  { ',' Parameter }

Parameter    ::=  Identifier  |  Structure

Structure    ::=  Identifier  { ':' Identifier }

               |  '<' Parameter_List '>'

Expression   ::=  Simple_Expression
                      { Infix_Operator Simple_Expression }

               |  'if' Expression 'then' Expression
                  'else' Expression

Simple_Expression  ::=  Application    |   Character
                        |  Construction  |   Identifier
                        |  Number        |   String
                        |  Tuple         |   '(' Expression ')'

Application     ::=  Identifier '(' Argument_List ')'

Argument_List   ::=  Expression  { ',' Expression }

Construction    ::=  '[' Argument_List ']'

Tuple           ::=  '<>'  |  '<' Argument_List '>'

Where_Clause    ::=  'where' Definition  { 'and' Definition }
```

Appendix B. PREDEFINED FUNCTIONS

This appendix catalogs the reserved words, predefined values, and predefined functions in Ernest.

Reserved Words

```
and       -- separates subordinate definitions
define    -- introduces a global function definition
div       -- infix operator for integer division
else      -- separates part of a conditional expression
if        -- introduces a conditional expression
o         -- infix operator for function composition
mod       -- infix operator for integer remainder
then      -- separates part of a conditional expression
where     -- introduces subordinate definitions
```

Predefined Values

```
false     -- Boolean negative
input     -- the input (keyboard) character stream
nil       -- the empty sequence
true      -- Boolean affirmative
undefined -- the value with no definition
```

Prefix Operators

```
-         -- arithmetic negation
~         -- logical complement (not)
```

Infix Operators

```
&         -- logical conjunction (and)
*         -- multiplication
+         -- addition
-         -- subtraction
/         -- division
/=        -- not equal
:         -- sequence construction (cons)
<         -- less than
<=        -- less than or equal
=         -- equal
>         -- greater than
>=        -- greater than or equal
^         -- sequence concatenation
div       -- integer division
mod       -- integer remainder
o         -- function composition
|         -- logical disjunction (or)
```

Predefined Functions

```
add(x,y)          -- addition (x+y)
chr(n)            -- numeric code to character conversion
cons(x,s)         -- sequence construction (x:s)
construct(s,x)    -- function construction
eager(f,x)        -- forces eager evaluation of x
equal(x,y)        -- test for equal values
filter(p,s)       -- subsequence extraction
generate(f,a)     -- sequence generation
grtr(x,y)         -- test for greater than (x>y)
grtreq(x,y)       -- test for greater than or equal (x>=y)
head(s)           -- first element of a sequence
less(x,y)         -- test for less than (x<y)
lesseq(x,y)       -- test for less than or equal (x<=y)
map(f,s)          -- sequence mapping
mult(x,y)         -- multiplication (x*y)
neg(x)            -- numeric negative
not(x)            -- logical negative
noteq(x,y)        -- test for inequality (x/=y)
ord(c)            -- character to numeric code conversion
or(x,y)           -- logical disjunction (x|y)
reduce(f,a,s)     -- sequence reduction
scan(f,a,s)       -- sequence scanning operation
sub(x,y)          -- subtraction (x-y)
tail(s)           -- sequence remainder
```

Appendix C. FUNCTION-FORMING OPERATIONS

In this appendix we present applicative definitions for the built-in function-forming operations described in Section 2.6. As is typical in interpreters, the built-in operations are much faster than the equivalent functions described here. In fact, functions defined using these building blocks are often faster than recursive applicative functions.

Construct

```
define  construct(s,x) = if s=<> then <>
                         else f(x) : construct(t,x)

        where  f:t = s
```

applies a tuple of functions to a common argument, producing the tuple of results: < f1(x), f2(x), ... >.

Filter

```
define  filter(p,s) = if s=<> then <>
                      else if p(x) then x:y
                      else y

        where  y = filter(p,t)

        and  x:t = s
```

selects the elements of a sequence that satisfy a given predicate.

Generate

```
define  generate(f,x) = x : generate(f,f(x))
```

applies a function repeatedly to produce the infinite sequence: < x, f(x), f(f(x)), ... >.

Map

```
define  map(f,s) = if s=<> then <>
                   else f(x) : map(f,t)

        where  x:t = s
```

applies a function to each element of a sequence, producing the sequence: < f(x1), f(x2), ... >.

Reduce

```
define  reduce(f,a,s) = if s=<> then a
                        else reduce(f,f(a,x),t)

        where  x:t = s
```

accumulates the result of applying a binary function "between" sequence elements.

Scan

```
define  scan(f,a,s) = a : if s=<> then <>
                          else scan(f,f(a,x),t)

        where  x:t = s
```

applies a binary function "between" sequence elements and produces a sequence of all the partial results.

Appendix D. COMPILER OPERATION

This appendix describes how to use Oliver, the compiler, and Sherbert, the run-time interpreter, to compile and run functional programs.

Oliver, the Compiler

Oliver translates high-level function specifications into a low-level code which is understood by the run-time interpreter. There are two parts to Oliver: Oliver1, which parses Ernest source code and produces an intermediate abstract syntax tree, and Oliver2, which processes the abstract syntax tree and generates low-level object code. The two commands necessary to compile an Ernest program called "example" are:

```
A> oliver1 example.fun example.ast

A> oliver2 example.ast example.hbc
```

The file extension "fun" is self-explanatory; "ast" stands for the abstract syntax tree; and "hbc" forms the initials of the developer of the theory behind the low-level code, Haskel B. Curry.

Both Oliver1 and Oliver2 read an additional file called OLIVER1.TBL and OLIVER2.TBL, respectively, which contains their translation tables. These files must reside in the default disk directory so the programs can find them.

Sherbert, the Interpreter

Sherbert reads and executes the low-level object code produced by the compiler. The command necessary to run our example functional program is:

```
A> sherbert example.hbc
```

Application programs can receive input from the keyboard. Sherbert reads this data and passes it to the application program through a predefined character string named "input." Examples of such programs are given in Section 3. Keyboard input can be terminated by typing a control-Z followed by a carriage return. All of Sherbert's output presently goes to the user's screen.

The batch command file included on the distribution diskette will compile a functional program and then execute the object code. It can be invoked by:

```
A> fungo example
```

Appendix E. ERROR MESSAGES

In this appendix we catalog the error messages that may be produced by our compiler and run-time interpreter. (At least these are the ones we have seen!)

Oliver's Error Messages

The compiler generates error and warning messages when it finds what it believes to be syntactically incorrect programs. This happens when the parser, following the grammar, cannot find a production to apply that will match the input. The actual error may be the current symbol that cannot be matched or an earlier error that led the parser to the wrong production.

In a few very simple cases, the compiler may try to correct the problem itself and continue the translation. When this occurs, a message such as the following is generated:

{line #} -- ")" inserted before {symbol}

Often, the compiler will produce a more ominous error message and then skip to the next convenient keyword or symbol and attempt to continue processing. These messages look like:

{line #} -- {symbol} unexpectedly encountered.
skipping to {symbol} in line {line #}

If the compiler can not get itself synchronized with recognizable input, it may give up with a message such as:

{line #} -- {symbol} found. Translation terminated.

or

{line #} -- {symbol} unexpectedly encountered. Translation terminated.

This information is not in a very friendly form but it usually provides enough of a clue to find obvious errors.

Sherbert's Error Messages

The runtime interpreter makes a number of checks while loading object programs, during program execution, and during garbage collection. Loading errors are usually caused by trying to load source code instead of object code. One of the following messages may be generated:

*** Loader -- error in number ***

*** Loader -- Unrecognized symbol {symbol} ***

*** Loader -- error in expression syntax ***

Many execution errors stem from incorrect use of recursion (which is a good reason to use function-forming operations!). These messages include:

```
*** Sherbert -- head applied to empty sequence ***
*** Sherbert -- tail applied to empty sequence ***
*** Sherbert -- forward cell on stack ***
*** Sherbert -- Unprintable expression on stack ***
*** Sherbert -- stack overflow ***
```

Larger programs may consume more storage space than is available. This will result in the following message during garbage collection:

```
*** Sherbert -- survivor space exhausted ***
```

Pascal Error Messages

As a final insult, it is possible for some errors to produce messages from the Pascal run-time system. The most common of these are caused by misspelling the program file name and by an arithmetic overflow. These messages look like:

```
? Error: File not found in file example
  Error Code 1032, Status 0002
PC = 0004: 109F; SS = 13BE, FP = 092D, SP = 6D5A
```

and

```
? Error: signed math overflow
  Error Code 2054
PC = 1346: 00B3; SS = 13BE, FP = 6F42, SP = 6F44
```

INDEX

Annex B. VAN USER'S GUIDE

VAN

DATA FLOW DIAGRAM EDITOR USER'S GUIDE

Version *i*

July 1986

Van

Data Flow Diagram Editor

User's Guide

Version i

July 1986

Audrey M. Rogerson
Reginald N. Meeson, Jr.

Computer Technology Associates, Inc.
7501 Forbes Blvd. Suite 201
Lanham, MD 20706
301-464-5300

This research was sponsored by the US Navy, Office of Naval Research under contract number N00014-84-C-0696.

Copyright (c) 1986, Computer Technology Associates, Inc.

Permission to copy this document for personal or educational use is granted provided that copies are not made or distributed for commercial advantage.

TABLE OF CONTENTS

PREFACE

This manual is a guide to the use of a data flow diagram programming system that contains an mouse-driven data flow diagram editor, called Van, and a compiler that translates data flow diagrams into executable code, called Vincent.

Van and Vincent were developed as part of a Small Business Innovative Research (SBIR) project sponsored by the U.S. Navy, Office of Naval Research.

Van and Vincent run on IBM PC's and compatible microcomputers. Van currently requires a Hercules graphics board. Van supports three kinds of mice (Mouse Systems, Summagraphics, and Microsoft) and produces hardcopy drawings on Epson-style dot-matrix printers. Vincent requires no special equipment and produces portable run-time code.

We would appreciate your comments on both the content and construction of this manual and on the tool itself. We will attempt to incorporate all viable suggestions as we continue to develop and enhance this product.

1. INTRODUCTION

This manual describes the operation of Van, a mouse-driven editor with which you can interactively create and modify data flow diagrams. Van allows you to manipulate data flow diagrams on the screen, save and retrieve diagrams, and print hard copies of the diagrams.

Van's data flow diagrams consist of the standard data flow elements (processes, interfaces, data stores, and flows) with an additional element which we call data flow diagram text. For a further discussion of data flow diagram techniques and conventions, see Tom DeMarco's Structured Analysis and System Specification (Prentice-Hall, 1979).

Section 2 describes how to use Van. It includes basic instructions on how to select commands and how to respond to prompts.

Section 3 provides a detailed description of the action and use of each editing command.

Section 4 demonstrates how to create data flow diagrams for a simple program. The problem and the diagrams that specify the solution are presented first. Then, an editing session that would create these diagrams is illustrated in detail.

Data flow diagrams created using Van can be compiled into executable code. Hence, you can program with data flow diagrams. An explanation of how to compile and execute a data flow diagram is given in Section 5. This section also includes a discussion of the semantics of data flow diagrams that are essential for compilation.

2. USING VAN

Getting Started

Van resides in a file called "VAN.EXE" and is activated by the command line

A> van

Van will first prompt you to indicate the type of mouse you have. Once Van receives this information, the initial menu is displayed with a blank screen so that the editing process can begin.

You interact with Van by means of the mouse and the keyboard. Commands, data flow elements, and positions on the screen are selected using the mouse. As you move the mouse, the position of the crosshair cursor changes on the screen. If the mouse does not move the cursor, then Van is expecting input from the keyboard. The keyboard is used for entering and editing textual information, such as file names and identifiers for data flow elements.

Selecting Commands

To select a command from the menu, move the crosshair cursor into the menu area. Each command is associated with a field that encloses the command name. As you move the cursor into a command field, the command will be highlighted. Then, pressing any button on the mouse will invoke the highlighted command. The command field will remain highlighted until the command is completed or cancelled. Figure 2-1 shows the position of the cursor as a command is selected. The details of what each command does are covered in the next section.

Selecting Locations and Elements

To select a location on the screen, move the crosshair cursor to the desired point and press any button on the mouse.

To select an element on the screen, move the crosshair cursor to that element and press any button on the mouse. For processes and data stores, place the cursor anywhere within the circle or box. To select an interface or a flow , you place the cursor in the vicinity of the center of the interface label or midpoint of the flow. By "in the vicinity" of a point we mean that the crosshair cursor extends out to that point. Finally, to select data flow diagram text on the screen, you can place the cursor anywhere on the text.

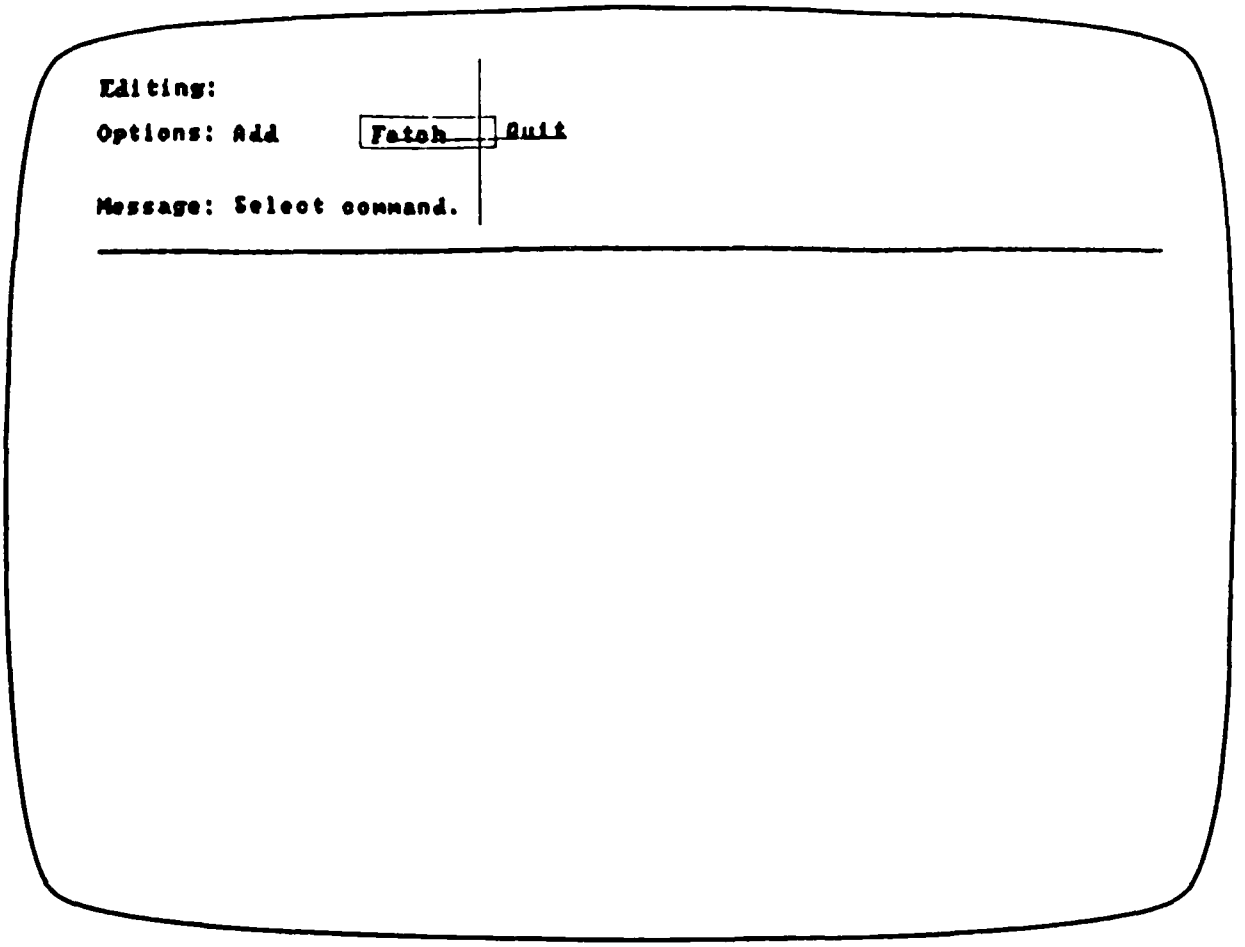


Figure 2-1. Positioning the Cursor to Select the Fetch Command

Try not to place elements in the diagram too close together. One reason for spacing out the elements is simply to make the diagram easier to understand. Van may have difficulty selecting the right element if elements overlap.

Editing Text

Entering and editing textual information, such as file names and identifiers on data flow elements, must be done from the keyboard. Van recognizes all of the alpha-numeric keys (upper and lower case) in addition to the following special keys:

Backspace -- moves the cursor to the left one character and deletes the character over it.

Back Tab -- moves the cursor to the beginning of the text.

Delete -- deletes the character over the cursor.

Enter	-- enters the text as it appears.
Escape	-- enters the original, unmodified text.
Insert	-- toggles text insertion mode.
Left Arrow	-- moves the cursor to the left one character.
Right Arrow	-- moves the cursor to the right one character.
Tab	-- moves the cursor to the end of the text.

Van's Menu

An example of a Van menu is illustrated in Figure 2-2. The menu is divided into three areas with the headings, "Editing", "Options", and "Message." The "Editing" area displays the name of the file which contains the current data flow diagram, the word "Modified" if the diagram has been changed, the word

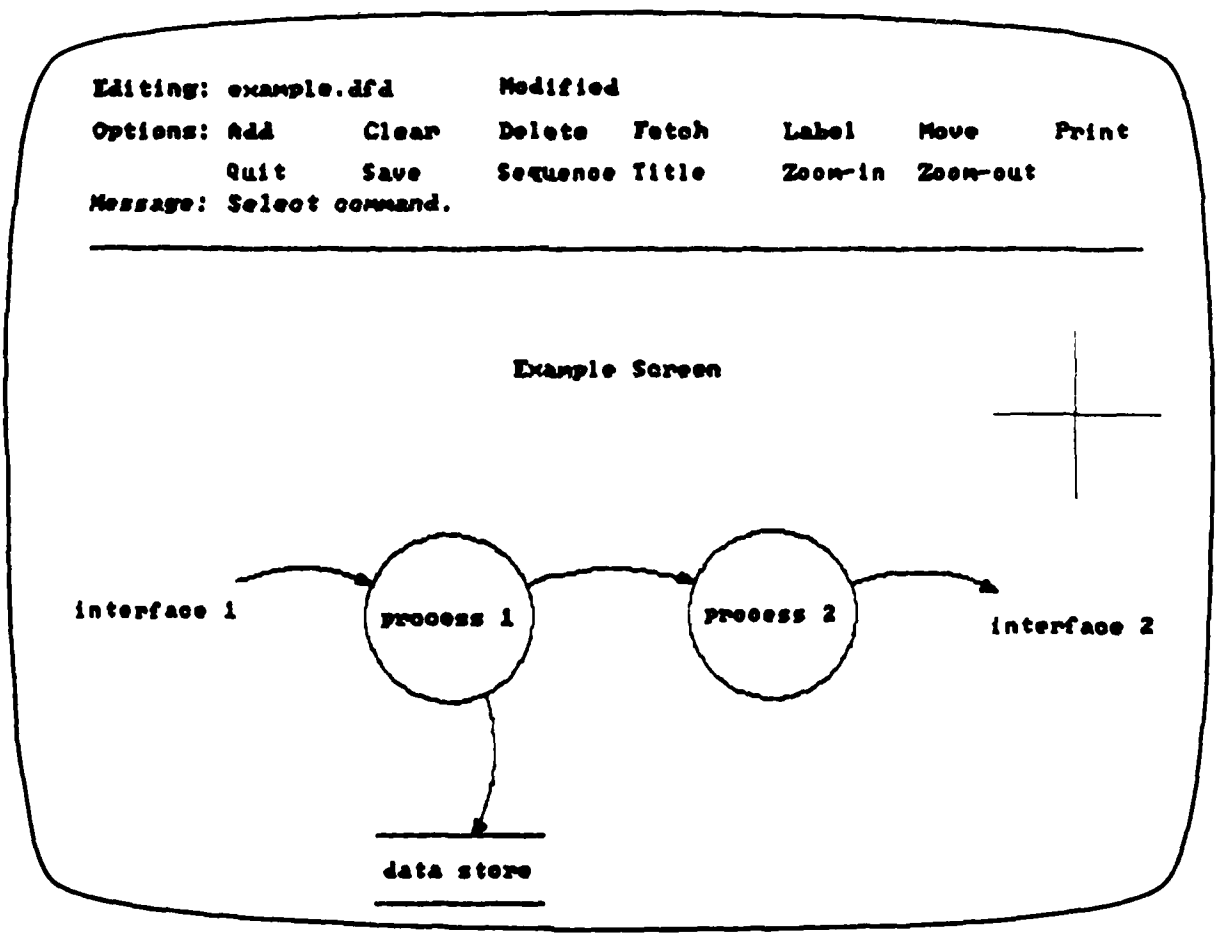


Figure 2-2. An Example Van Menu and Screen

"Saving" when the diagram is being saved, and the insert mode ("Insert On" or "Insert Off") when you are editing text.

The "Options" area displays the list of available commands. There are three different sets of available commands which correspond to different stages in the editing process. The first set is the list of Initial commands. This list appears in the beginning of the editing session when the screen is blank. The complete list of Van's commands is called the list of "Normal" commands. Van displays the list of "Normal" commands when the screen contains at least one data flow element. The third set is the list of "Add" commands which appears when you select the "Add" command from either of the other two lists. The three lists of commands are given below:

Initial commands:

Add	Fetch	Quit

Normal commands:

Add	Clear	Delete	Fetch	Label	Move	Print
Quit	Save	Sequence	Title	Zoom-in	Zoom-out	

Add commands:

Addflow	Addiface	Addproc	Addstore	Addtext	End

The "Message" area of the menu displays prompts and messages for the user. The "Message" area is also where labels and data flow diagram text are displayed for editing purposes.

3. THE EDITING COMMANDS

This section provides a detailed guide to the action and use of each editing command. In general, you can cancel a command which first prompts for input from the mouse by moving the cursor into the menu area and pressing any button on the mouse. Commands which first prompt for input from the keyboard can be cancelled by pressing the escape key, ESC. Prompts for a "yes" or "no" will be followed by "(Y/n)" or "(y/N)," where the default appears in upper case. You may respond by typing "Y" or "N" in either upper or lower case.

Add -- Add a data flow element to the current diagram.

When you select the "Add" command, Van displays the "Add" menu and enables you to select one of the following commands:

Addflow -- Add a data flow.

When you select the "Addflow" command, Van displays the prompt:

"Select the starting element for the flow"

in the message area of the menu. After you select the source element for the flow, Van displays the prompt:

"Select the destination element for the flow."

After you select the destination element Van draws the arrow and displays the prompt:

"Enter data flow name:"

for you to enter the name of the new data flow (which may be blank).

This command can be cancelled at any time before Van draws the arrow by moving the cursor into the menu area and pressing any button on the mouse.

Addiface -- Add an interface.

When you select the "Addiface" command, Van displays the prompt:

"Select the location of the interface."

Once you select the location, Van displays the prompt

"Enter interface name:"

When you enter the name of the interface, Van centers it at the location you selected.

This command can be cancelled before you select the location by moving the cursor into the menu area and pressing any button on the mouse.

Addproc -- Add a process.

When you select the **Addproc** command, Van prompts:

"Select the location of the process."

Once you select the location, Van draws a circle centered at that point. Van then displays the prompt:

"Enter process name:"

When you enter the name, Van centers it in the circle. Van then prompts:

"Enter source file name (filename.ext):"

This is where you enter the name of the file which contains (or will contain) the data flow diagram that defines this process. If there is no further definition then simply enter a carriage return.

This command can be cancelled before you select the location by moving the cursor into the menu area and pressing any button on the mouse.

Addstore -- Add a data store.

When you select the **Addstore** command, Van prompts:

"Select the location of the store."

After you select the location, Van draws a data store symbol centered at that point. Van then displays the prompt:

"Enter data store name:"

When you enter the name, Van centers it inside the data store symbol.

This command can be cancelled before you select the location by moving the cursor into the menu area and pressing any button on the mouse.

Addtext -- Add data flow diagram text.

When you select the **Addtext** command, Van prompts:

"Select the location for the beginning of the text."

Once you select the location, Van displays the prompt,

"Enter text:"

and writes the text beginning at the selected location.

This command can be cancelled before you select the location by moving the cursor into the menu area and pressing any button on the mouse.

Clear -- Clear the screen.

This command clears the screen so that a new data flow diagram can be created. If the current diagram has been modified when you select this command, Van will remind you to save the diagram by displaying,

"Save the current diagram? (Y/n)"

If you do not want to save the diagram, you must respond by typing "N". Otherwise, Van will prompt,

"Enter file name in which to save diagram (filename.ext): [current filename]"

Van displays the file name for the current diagram in case you want to edit it. Van will save the diagram, as directed, and clear the screen.

Delete -- Delete a data flow element from the current diagram.

When you select this command, Van prompts:

"Select element to delete."

When you have selected the element to delete, Van requests you to confirm the deletion by displaying the prompt:

"Delete [element type]? (y/N)"

You must type a "Y" to delete the element. Otherwise, the command is cancelled.

This command can also be cancelled before you select an element by moving the cursor into the menu area and pressing any button on the mouse.

When you select the element to delete, if the cursor is not close enough to the element Van will display the message:

"No element selected - try again"

End -- End the Add command.

When you no longer want to add elements to the diagram, this command will take you back to the menu with the list of normal commands.

Fetch -- Retrieve a saved diagram.

When you select this command, Van prompts:

"Save the current diagram? (Y/n)"

to remind you to save the current diagram first. Unless you type "N", the diagram will be saved. Van then displays the prompt:

"Enter the name of the file to retrieve (filename.ext):"

If the file cannot be found in the current directory, Van will display the message:

"Unable to find file. Check file name"

and cancel the command. Otherwise, Van will display the message:

"Reading display file"

Van will then clear the screen, and draw the new data flow diagram specified in the file. If the file does not contain a diagram readable to Van, the message:

"Format error in display file. Command cancelled"

will be displayed.

Label -- Add or Edit a data flow element label.

This command is used to edit labels on data flow elements, such as data flows, interfaces, processes, data stores, and also to edit text. When you select this command, Van displays the prompt:

"Select element to label."

When the element is selected, Van prompts for the new label

with

"Enter label: [current label]"

and allows you to edit the current label. If the element selected is a process, Van will also issue the following prompt after you have entered the new label:

"Enter source filename (filename.ext): [current filename]"

so that you can edit this if necessary. The new label will then replace the old one in the diagram. For processes, the new file name will also be stored.

This command can be cancelled before you select the element by moving the cursor into the menu area and pressing any button on the mouse.

Move -- Move a data flow element.

When you select this command, Van displays the prompt:

"Select element to move."

If the element selected is an interface, process, data store or text, Van displays the prompt:

"Select the new position for the [element type]."

The element type is displayed to help verify that the right element was selected. When you select the new position, Van redraws the diagram to reflect the change.

If the element selected is a data flow, Van issues the prompt:

"Select the new starting element for the flow."

When the selection is made, Van prompts:

"Select the new destination element for the flow."

When the second selection is made, Van redraws the diagram.

This command can be cancelled before you select the new position (in the case of moving a data flow, before you select the destination element) by moving the cursor into the menu area and pressing any button on the mouse.

Print -- Print the current data flow diagram.

When you select this command, Van displays the prompt:

"Print the full diagram? (Y/n)"

"Full" diagrams include extra information required for compilation (see Section 4). If you do not want this information to appear in the printed diagram, respond by typing an "N". This will cause Van to display the prompt,

"Print the basic diagram? (Y/n)"

"Basic" diagrams do not show any of the extra compilation information.

Van redraws the current data flow diagram to keep the printed copy in proper proportion. When the printing is completed, Van restores the original image on the screen.

Quit -- Quit the editing session. Exit Van.

When you select this command, Van will ask you to save the current diagram if it has been modified by issuing the prompt:

"Save the current diagram? (Y/n)"

You must type "N" to exit Van without saving the current diagram. Otherwise, Van will save the diagram and exit to the operating system.

Save -- Save the current data flow diagram.

When you select this command, Van displays the prompt:

"Enter file name in which to save diagram (filename.ext) [current filename]"

Van displays the current file name so that you can edit it if you want to save the current diagram in a new file. If you enter a carriage return without changing the file name, Van will overwrite the original file.

Sequence -- Sequence the flows into a process.

This command is used to specify the order of arguments for a process by sequencing the incoming flows. When you select this command, Van displays the prompt:

"Select the process to sequence"

and waits for a selection. When the selection is made, Van displays the prompt:

"Select the flows into this process in order."

Select the flows into the process in the order which you desire. As you make the selections, Van appends the labels on the flows with "([sequence number])." After you select the last flow, cancel the selection process by moving the cursor into the menu area and pressing any button on the mouse. The **Sequence** command itself can be cancelled at any time in the same manner.

Title -- Add or Edit the data flow diagram title.

When this command is selected, Van displays the prompt:

"Enter title: [current title]"

You can then edit the current title, if one exists, or create a new title. Van centers the title at the top of the working screen. Note that the title is not a data flow element and therefore cannot be moved or deleted using the "Move" or "Delete" commands. You can remove the title by selecting the "Title" command and blanking out the field.

Zoom-in -- Move down one level in the diagram.

The **Zoom-in** command is used to travel down the tree structure of the diagram. When you select this command, Van saves the current diagram if it has been modified and displays the prompt:

"Select the process to zoom in on."

If the process you select has not had its source file name specified, Van will display the message,

"Process doesn't have a function id -- command cancelled."

If the process has a source file name but Van is unable to locate the file, Van will display the message:

"Unable to find file. Check source file name"

and cancel the command. Otherwise, Van will read in the display file which represents the definition of that process and display the message:

"Reading display file"

Van will then clear the screen and draw the new data flow diagram specified in the file. If the file does not contain a diagram that Van can read, the message:

"Format error in display file. Command cancelled"

will be displayed.

This command can be cancelled, before you select the process to zoom in on, by moving the cursor into the menu area and pressing any button on the mouse.

Zoom-out -- Move up one level in the diagram.

This command is used to travel up the tree structure of the diagram. If the current diagram is the top-level diagram when you select the Zoom-out command, Van will display the message:

"At top level. You cannot zoomout"

and cancel the command. Otherwise, Van will save the current diagram if it has been modified and display the message:

"Reading display file"

Van will then clear the screen and draw the parent diagram.

4. A SAMPLE EDITING SESSION

In this section we demonstrate how Van is used to create data flow diagrams for a simple function definition. First we describe the example problem and its solution in the form of data flow diagrams. The we illustrate in detail the sequence of commands that generates these diagrams.

The Problem to be Solved

Consider the problem of averaging a sequence of integers. To average a sequence, "s", of integers, you divide the sum of the integers by the number of integers in the sequence. The data flow diagram which corresponds to this definition is shown in Figure 4-1.

We now need to define the sum function. We use the Ernest function-forming operation "reduce." "Reduce" accumulates the result of applying a binary function ("+") between elements in the sequence, s. The initial value of the accumulated result (0 for addition) must also be supplied. The function "reduce(f)", therefore, is the function that adds all the elements of "s" to the initial value 0. Using this function, we can easily represent sum(s) by the data flow diagram shown in Figure 4-2.

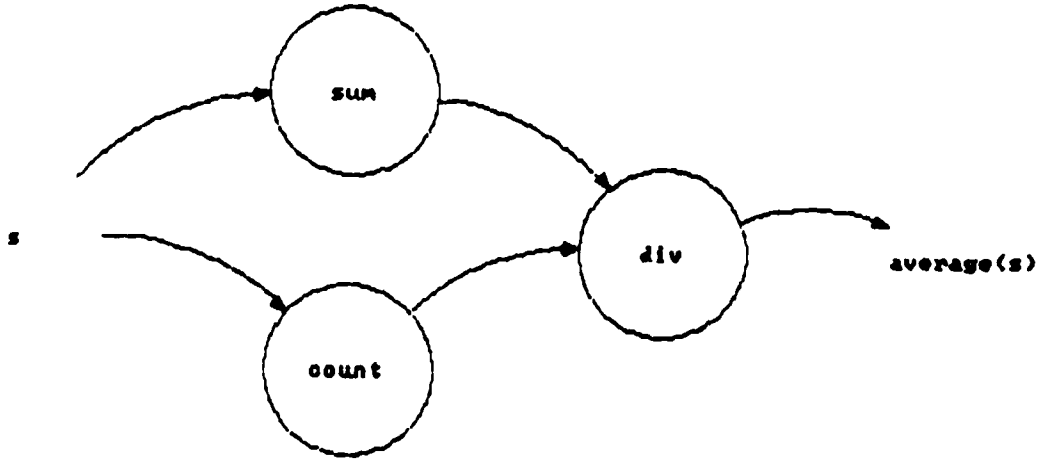


Figure 4-1. Data flow diagram for "average(s)"

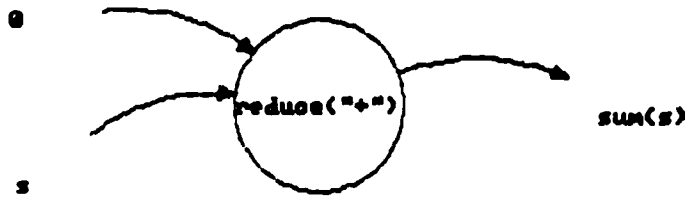


Figure 4-2. Data flow diagram for "sum(s)"

Similarly, we can represent "count(s)" with the diagram shown in Figure 4-3. In this case, though, the accumulation function, "incr", adds the value 1 for each element in "s" instead of the values of the elements. That is,

```
incr(a,x) = a + 1     -- for any element x
```

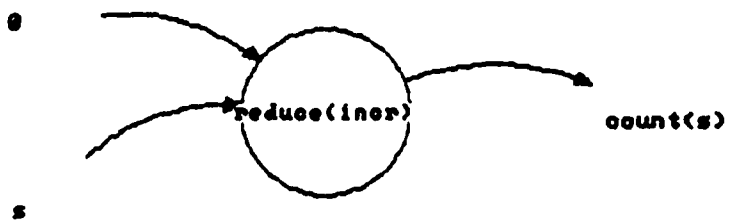


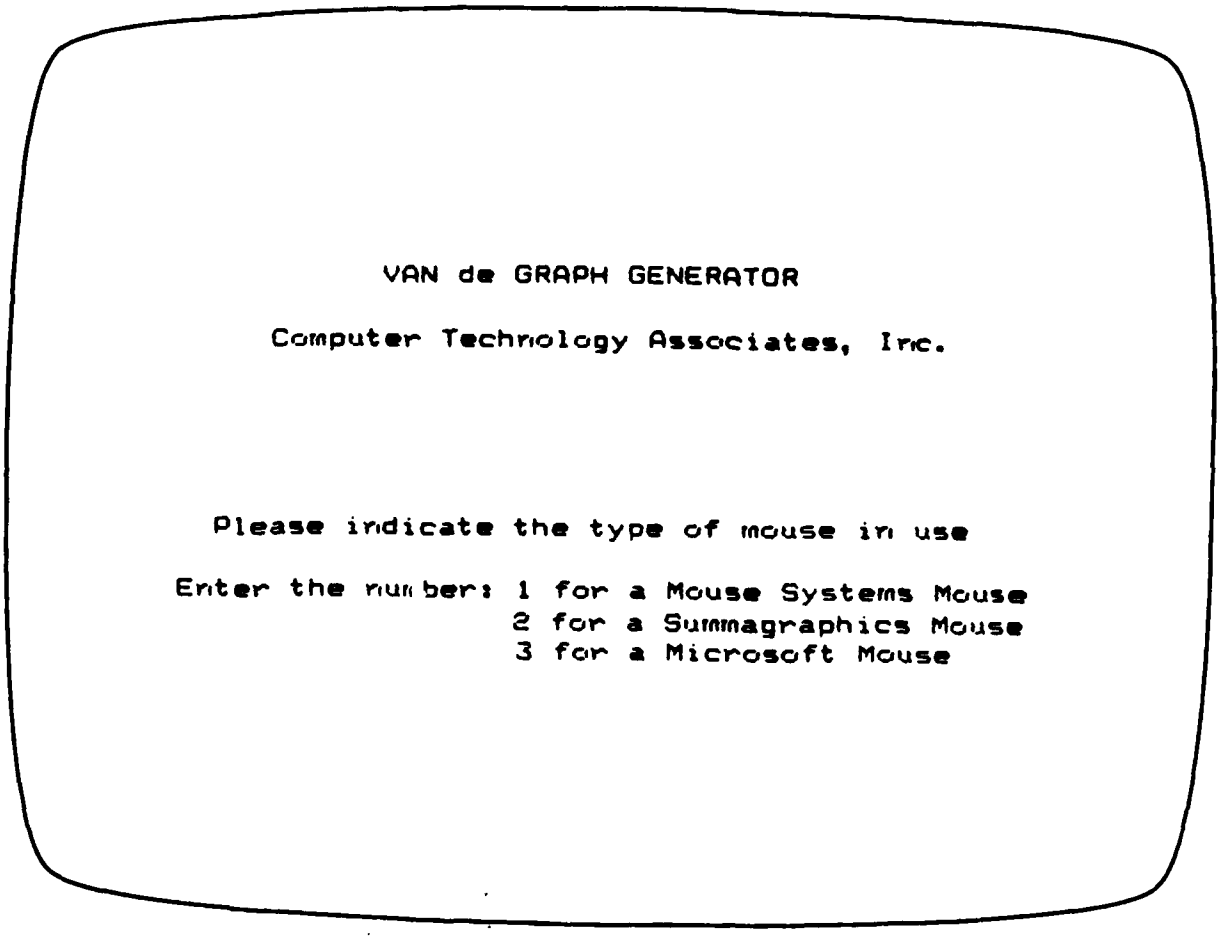
Figure 4-3. Data flow diagram for "count(s)"

Drawing the Diagrams

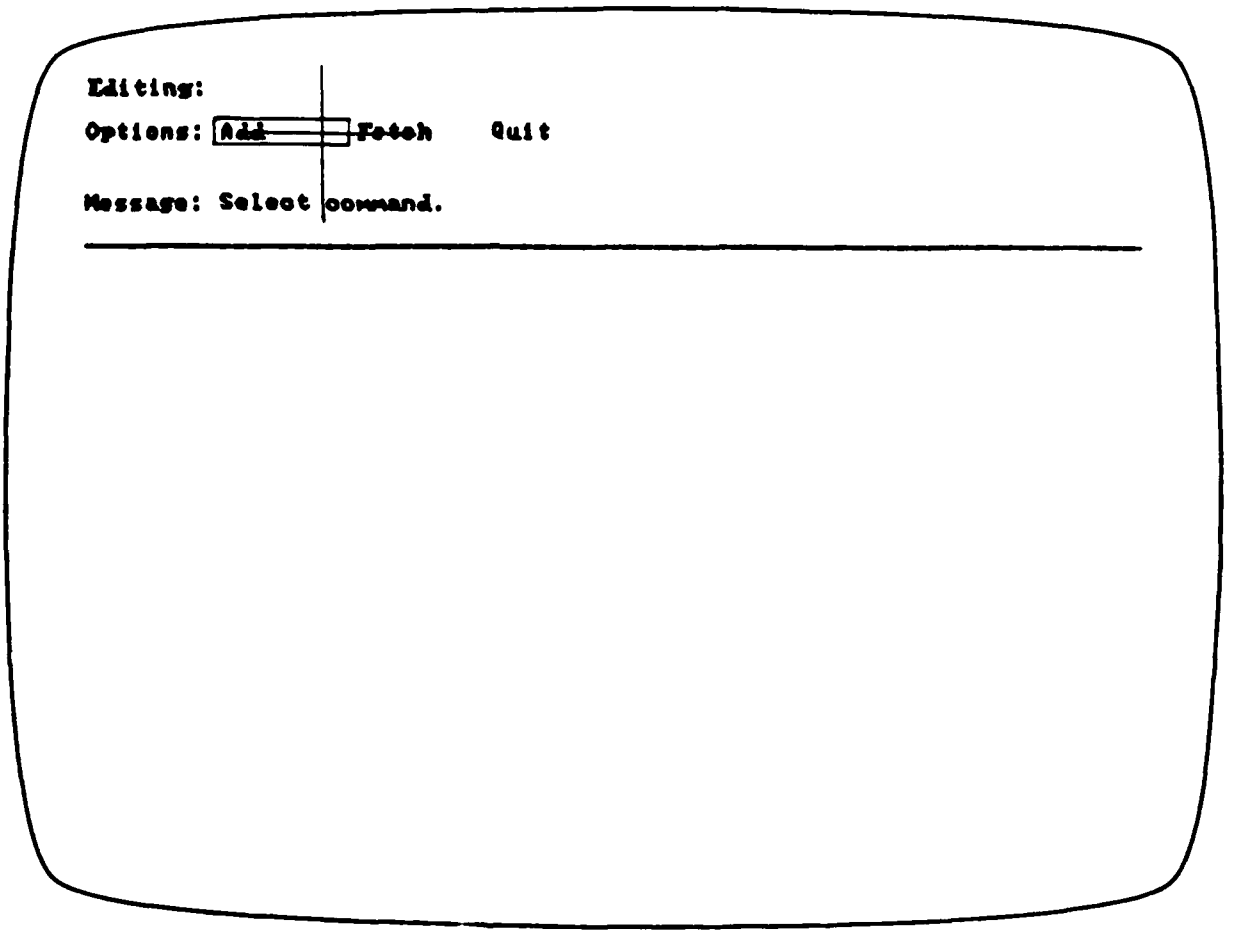
Once Van is activated by the command line,

A> van

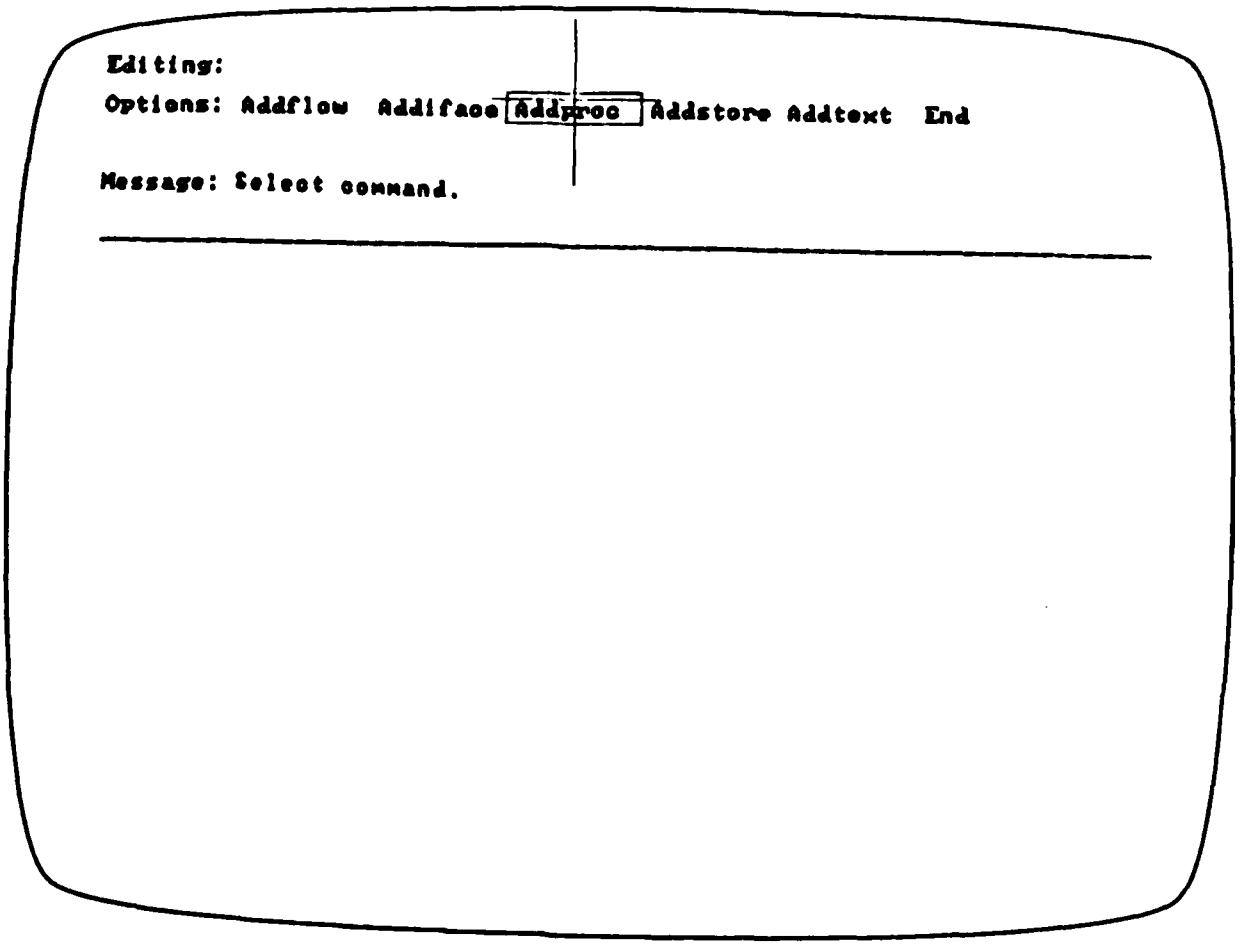
an introductory screen will appear. The introductory screen prompts you for the type of mouse you will be using. Enter the number which corresponds to your mouse.



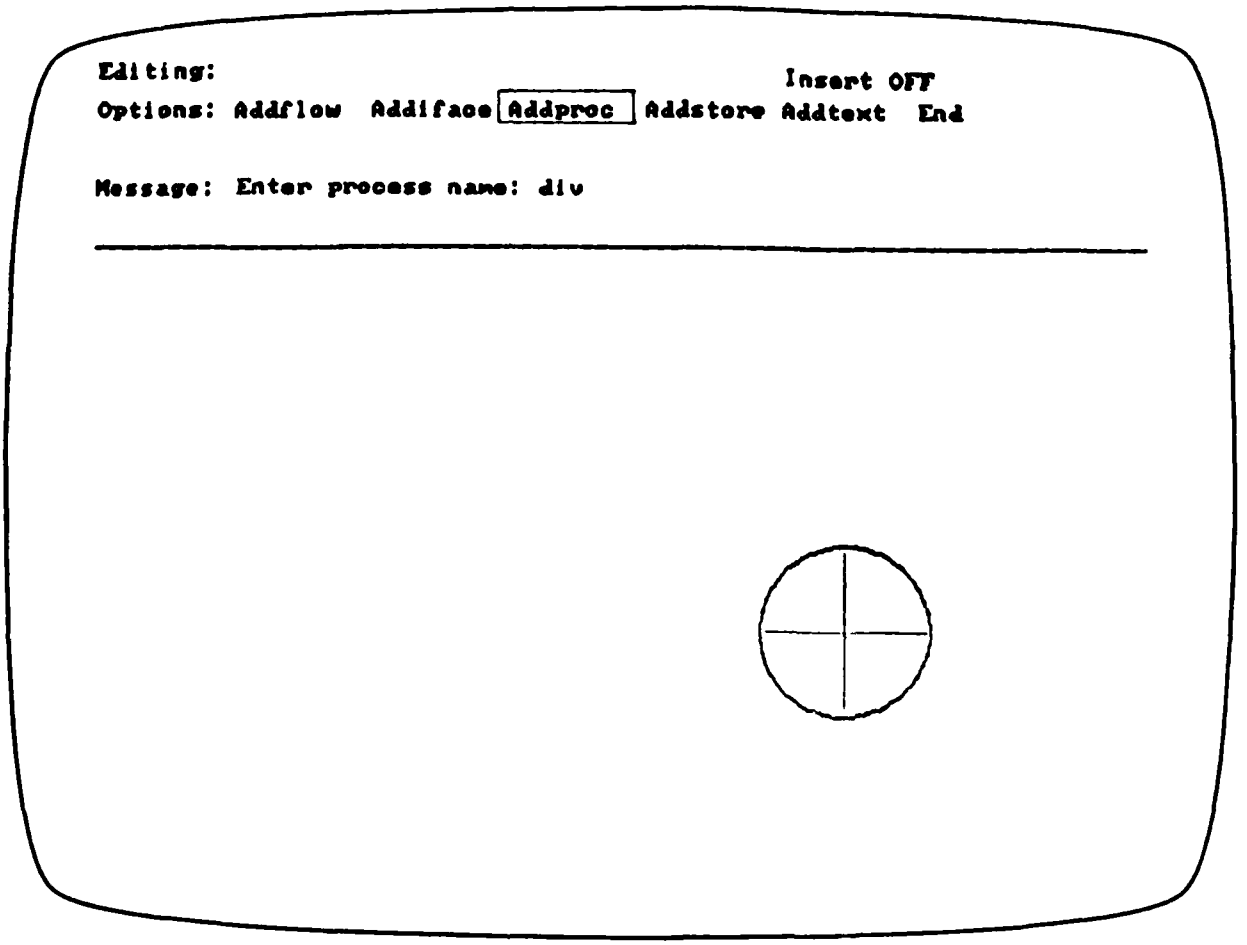
Van now displays the initial menu with a blank screen. If you want to the draw the diagram for average(s) first, select the "Add" command from the initial menu.



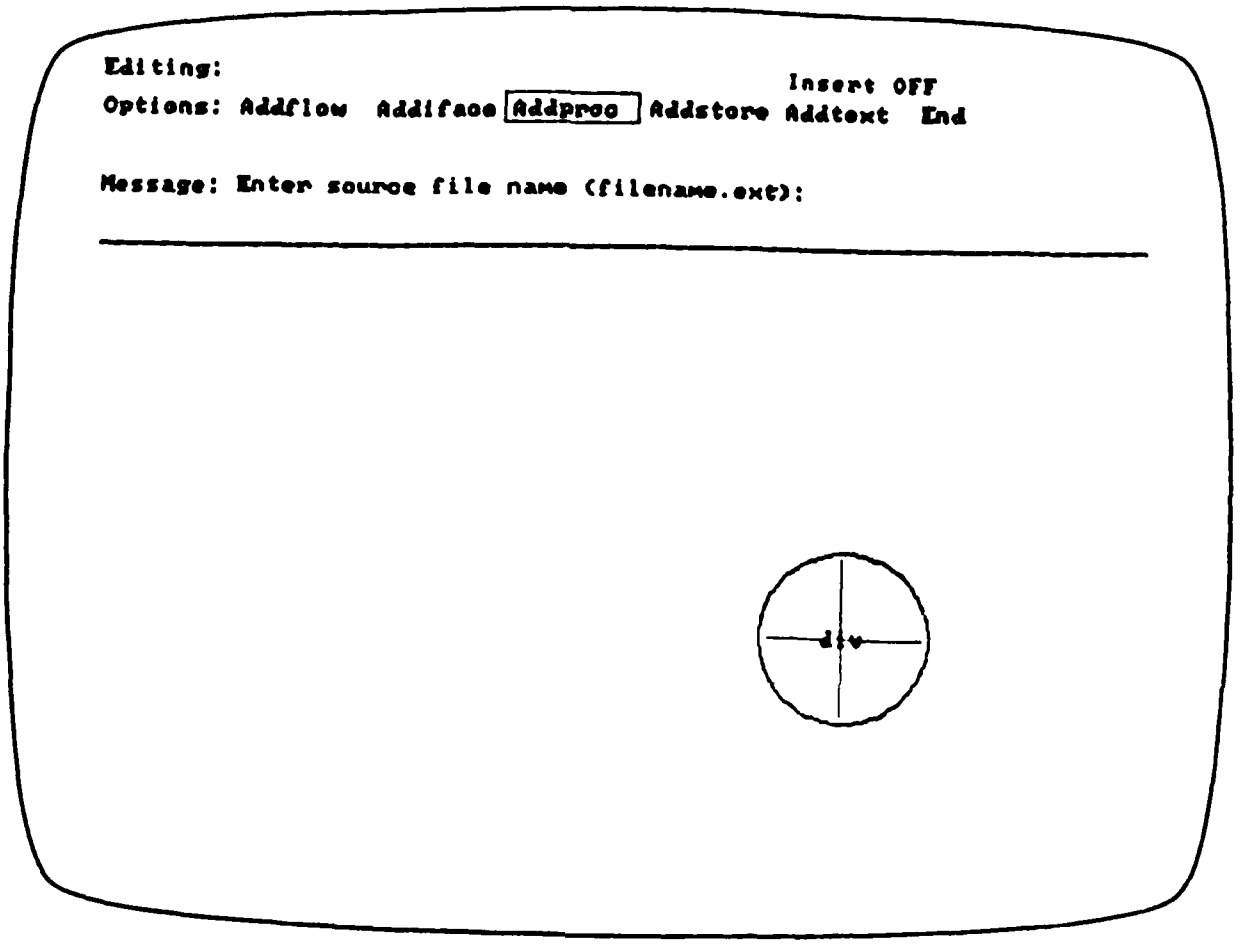
When Van displays the "Add" menu, select "Addproc" to draw the first process.



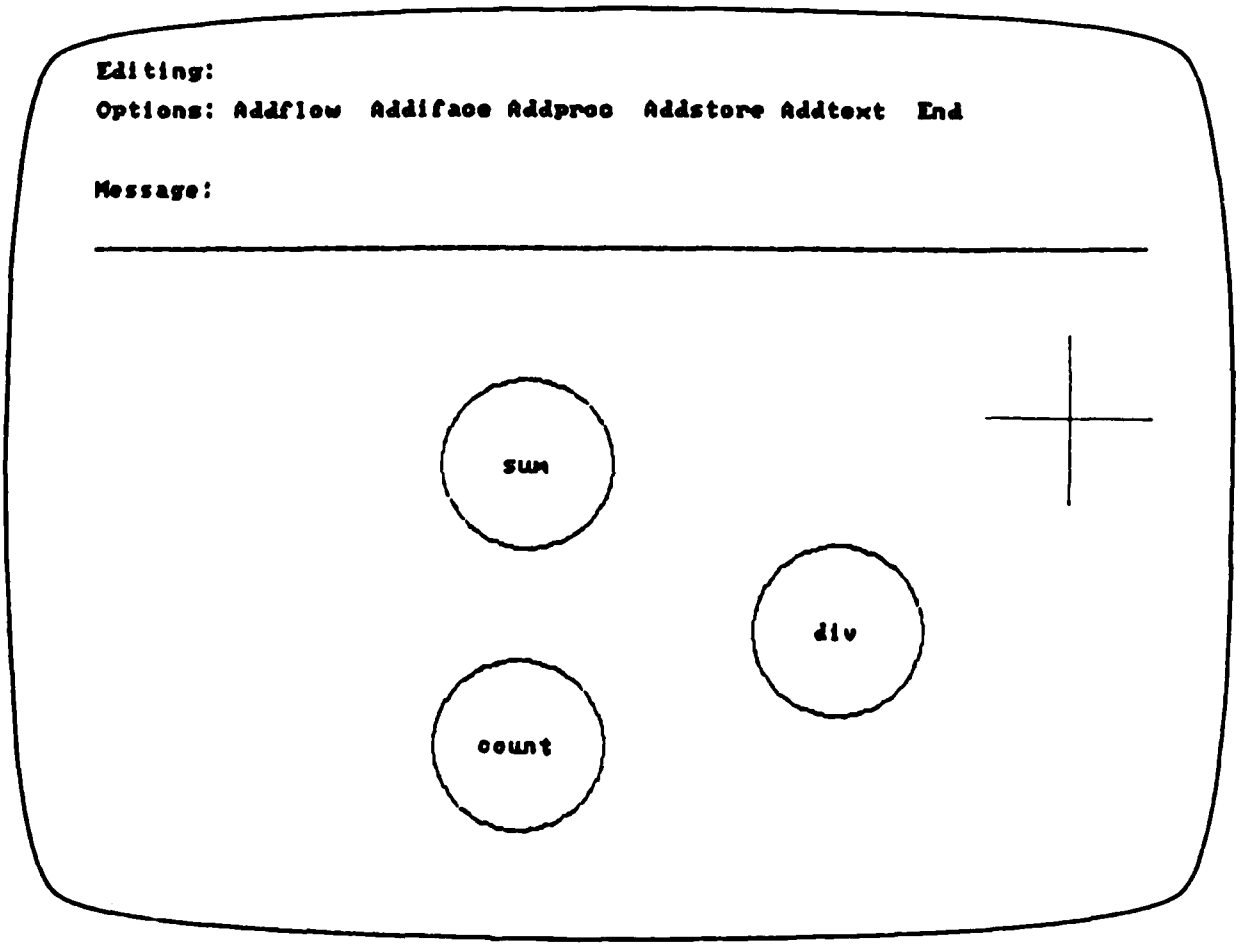
Van now prompts for the location of the process. When you select the location, Van draws the circle and prompts you for the name of the process. Key in the name of the process, "div".



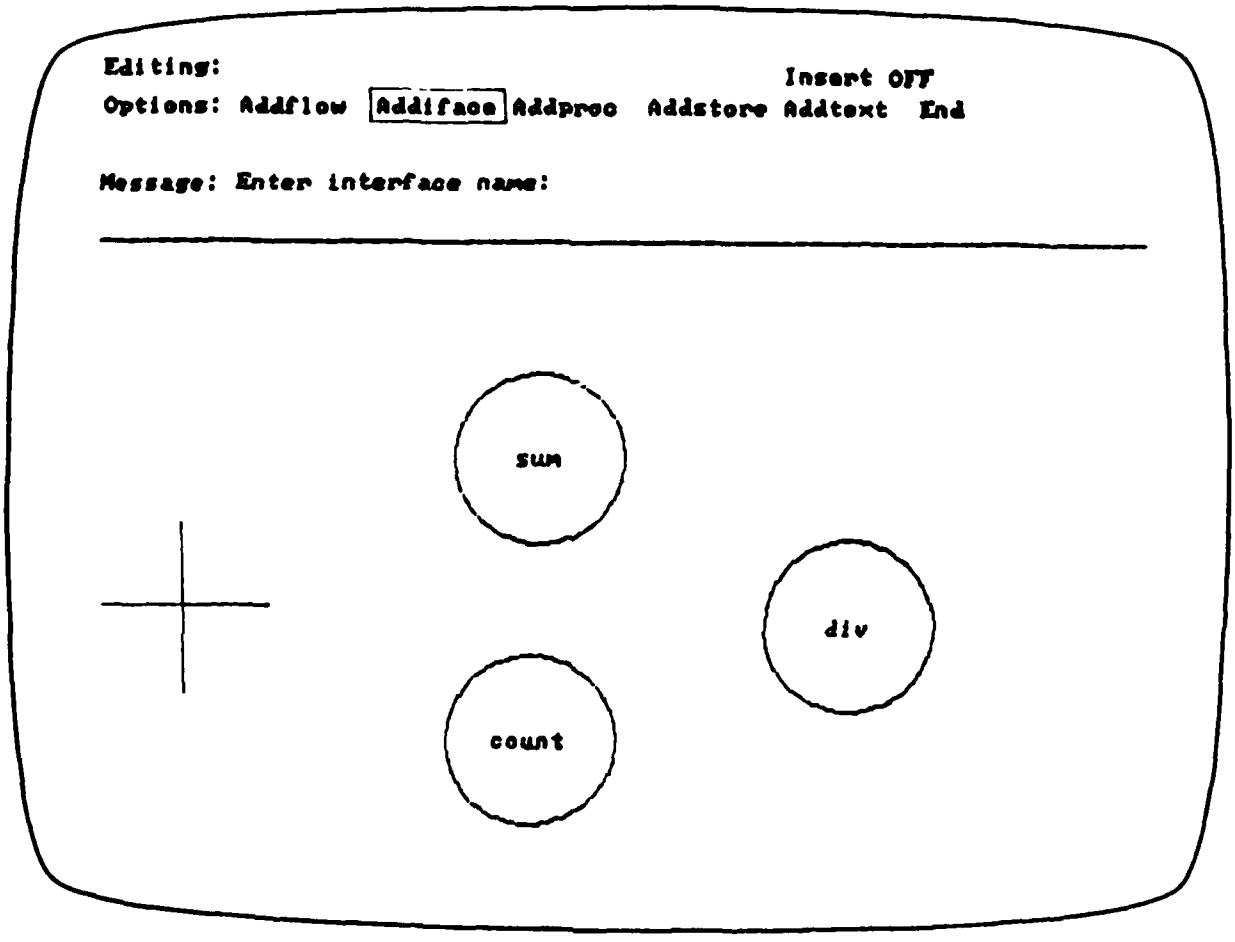
When you press the Return key, Van puts the name of the process in the circle and prompts for the source file name.



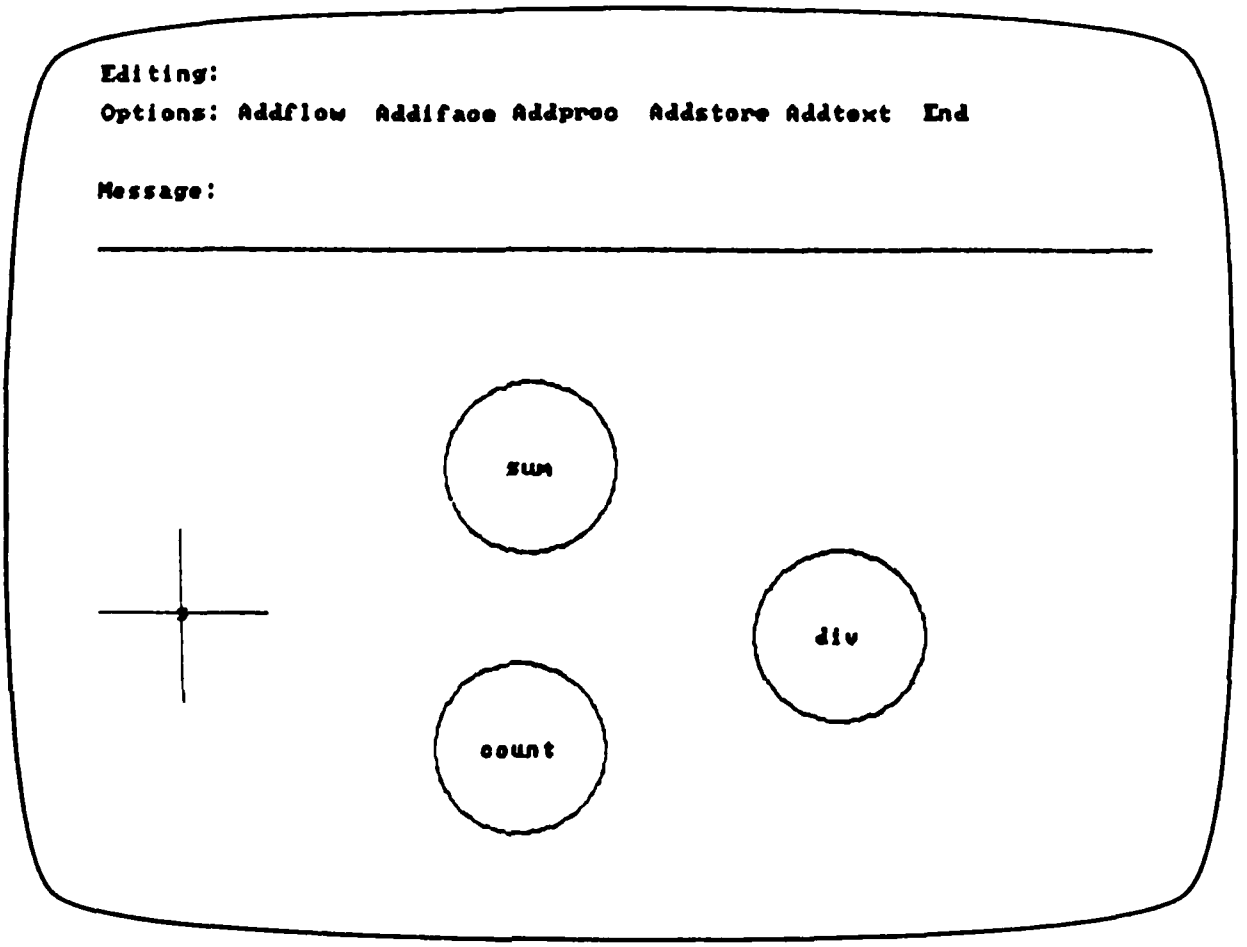
Since "div" is a built in function, we do not need to define it with a data flow diagram and can therefore answer the prompt for a source file name by pressing Return. You can continue in this manner until the three processes in the data flow diagram for "average(s)" have been added to the diagram. Note that the source file names for the processes "sum" and "count" can be specified when the processes are added, if you know what files they reside or will reside in, or later using the "Label" command.



If you then want to add the interfaces for the diagram, select the "Addiface" command. Van will prompt you to select the location of the interface. Select the location for, say, the input interface. Van then prompts for the name of the interface.

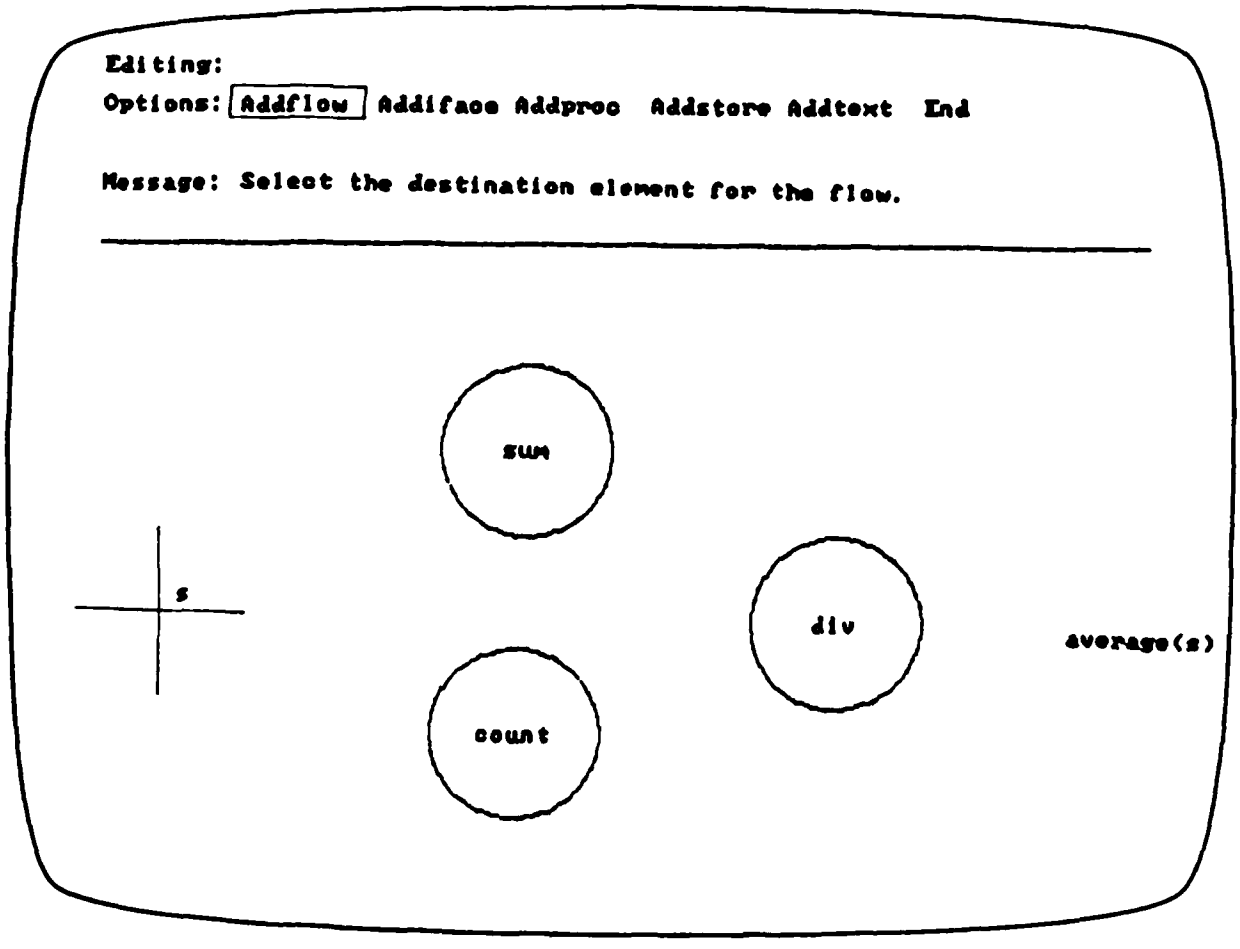


After responding to the prompt by entering "s", Van will draw the name "s" at the location and clear the prompt.

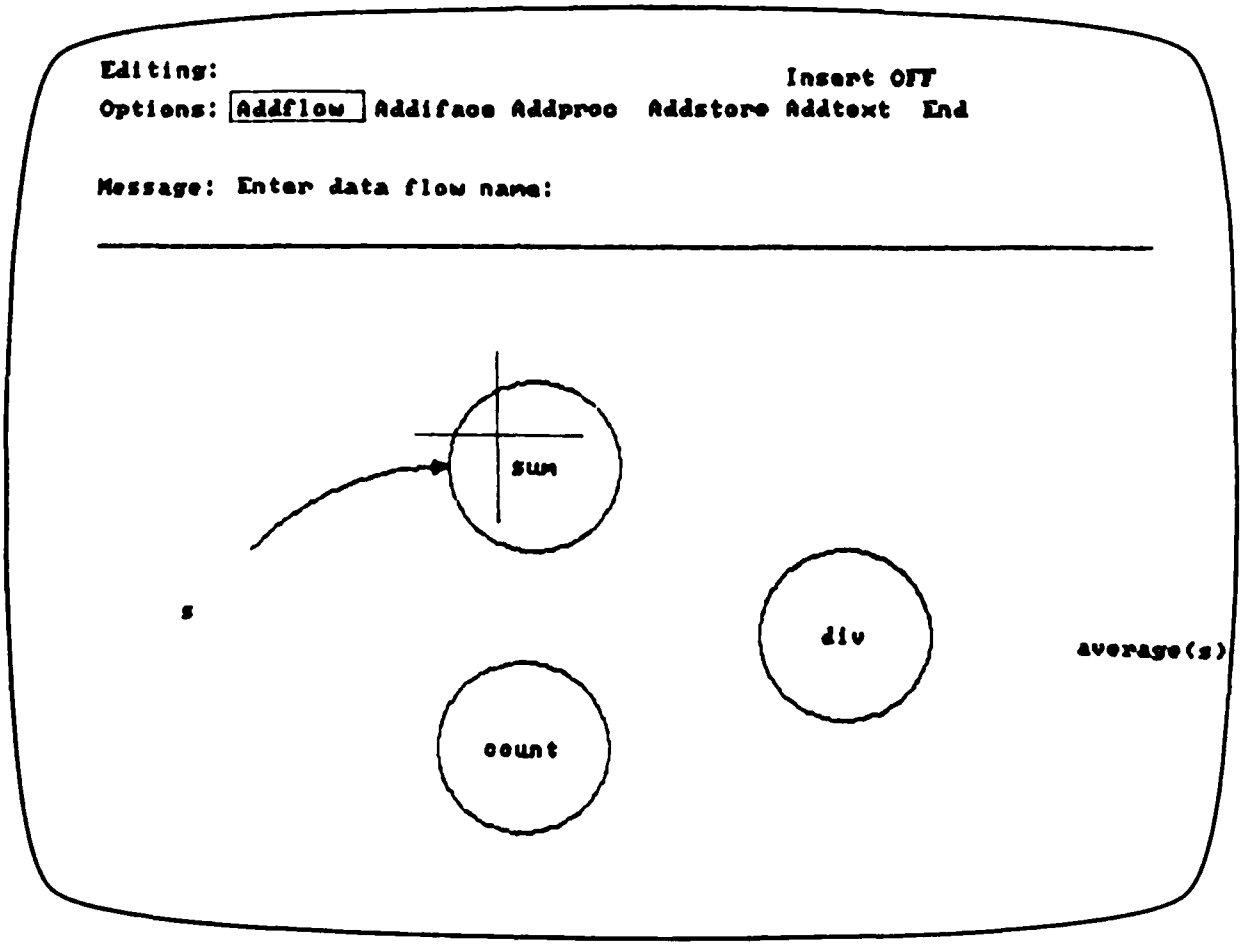


The output interface, "average(s)", is entered in the same manner.

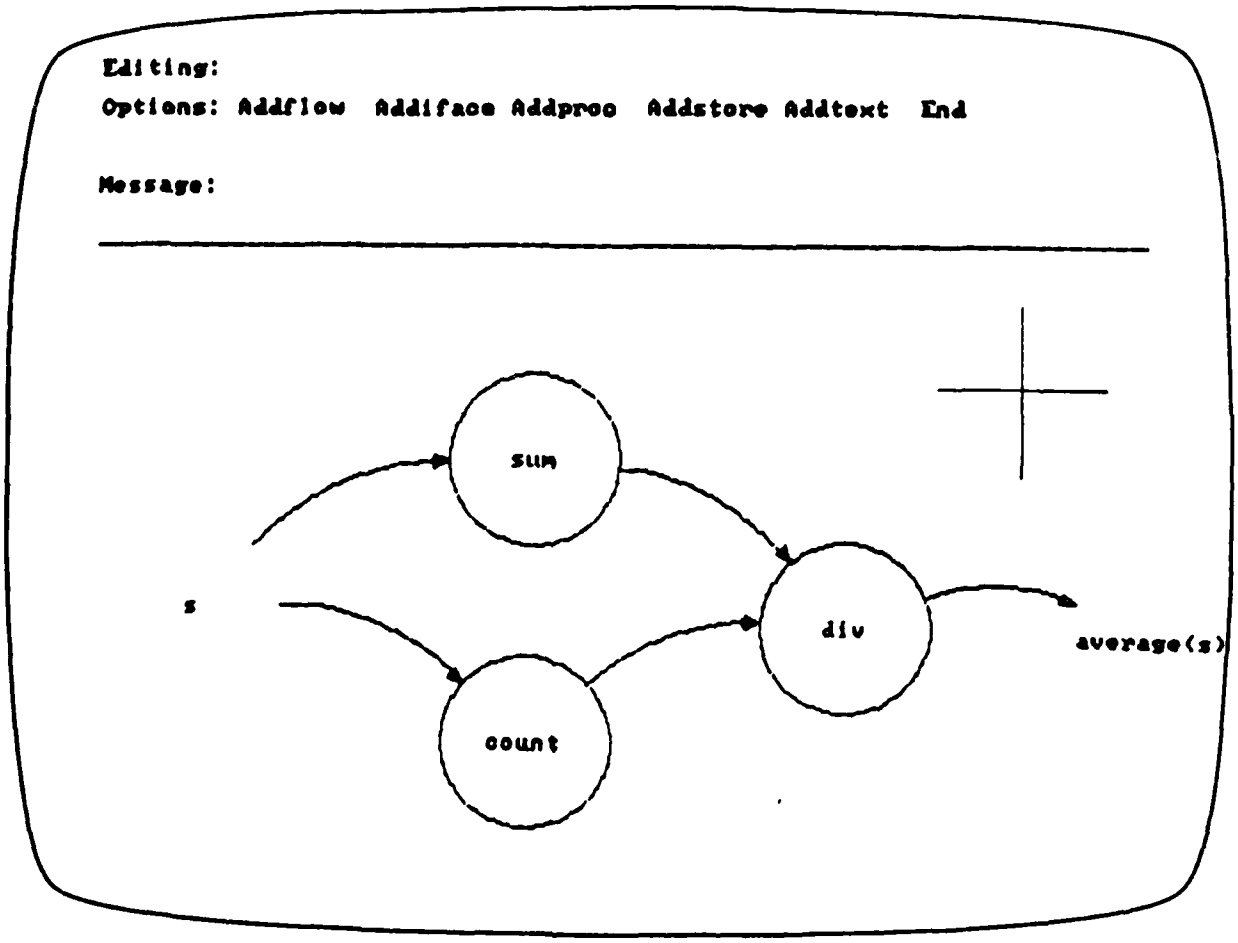
To add the flow from the input interface "s" to the process "sum", select the **Addflow** command. When Van prompts you for the starting element, select the interface "s." When this is selected, Van prompts you for the destination element.



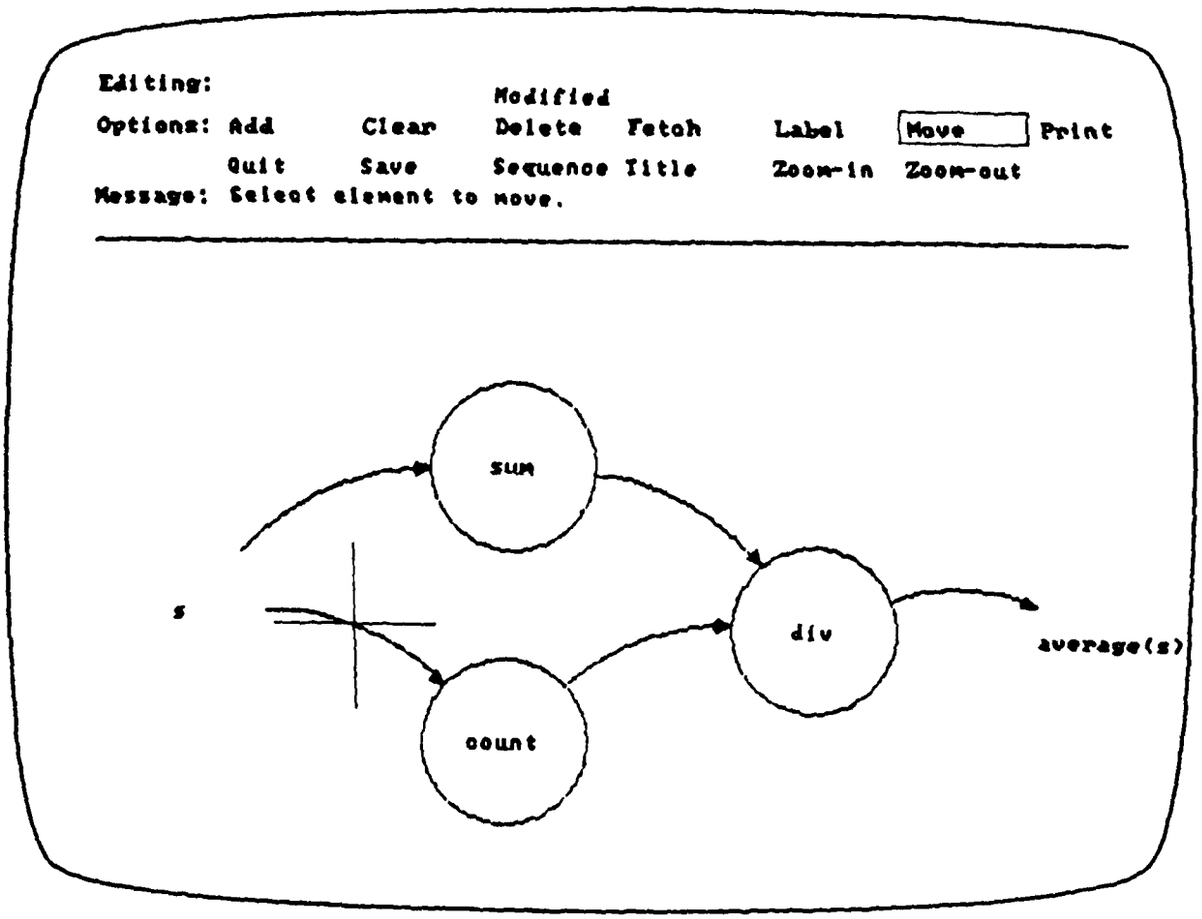
If, for example, you select the process "sum", Van draws the flow and prompts for the name of the flow.



Since this flow has no label, you can respond by pressing Return. This completes the **Addflow** command. The other flows in the diagram are added in the same manner.

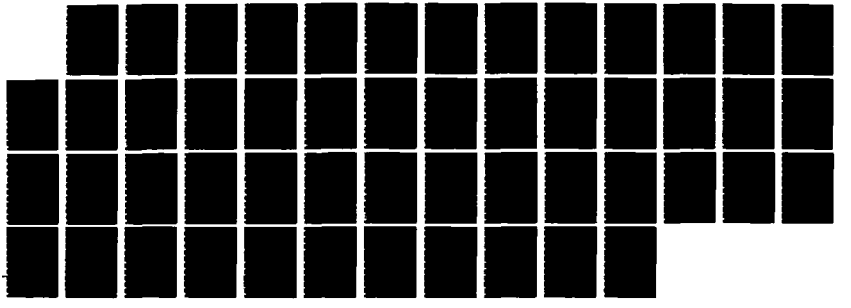


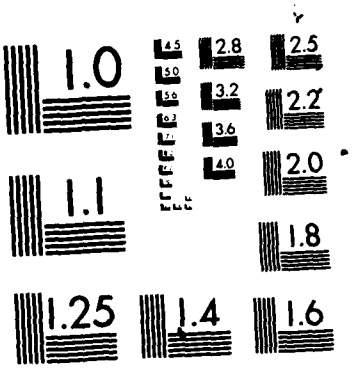
If you want to clean up the diagram by allowing only one flow from the input interface "s", **End** the **Add** command and select the **Move** command. Once you select the **Move** command, Van prompts you to select the element to be moved. Select one of the flows originating at the interface "s."



AD-A172 958 FUNCTIONAL PROGRAMMING(U) COMPUTER TECHNOLOGY 2/2
ASSOCIATES INC LANHAM MD R N MEESON AUG 86
CTA-031-3017001-8602A N00014-84-C-0696

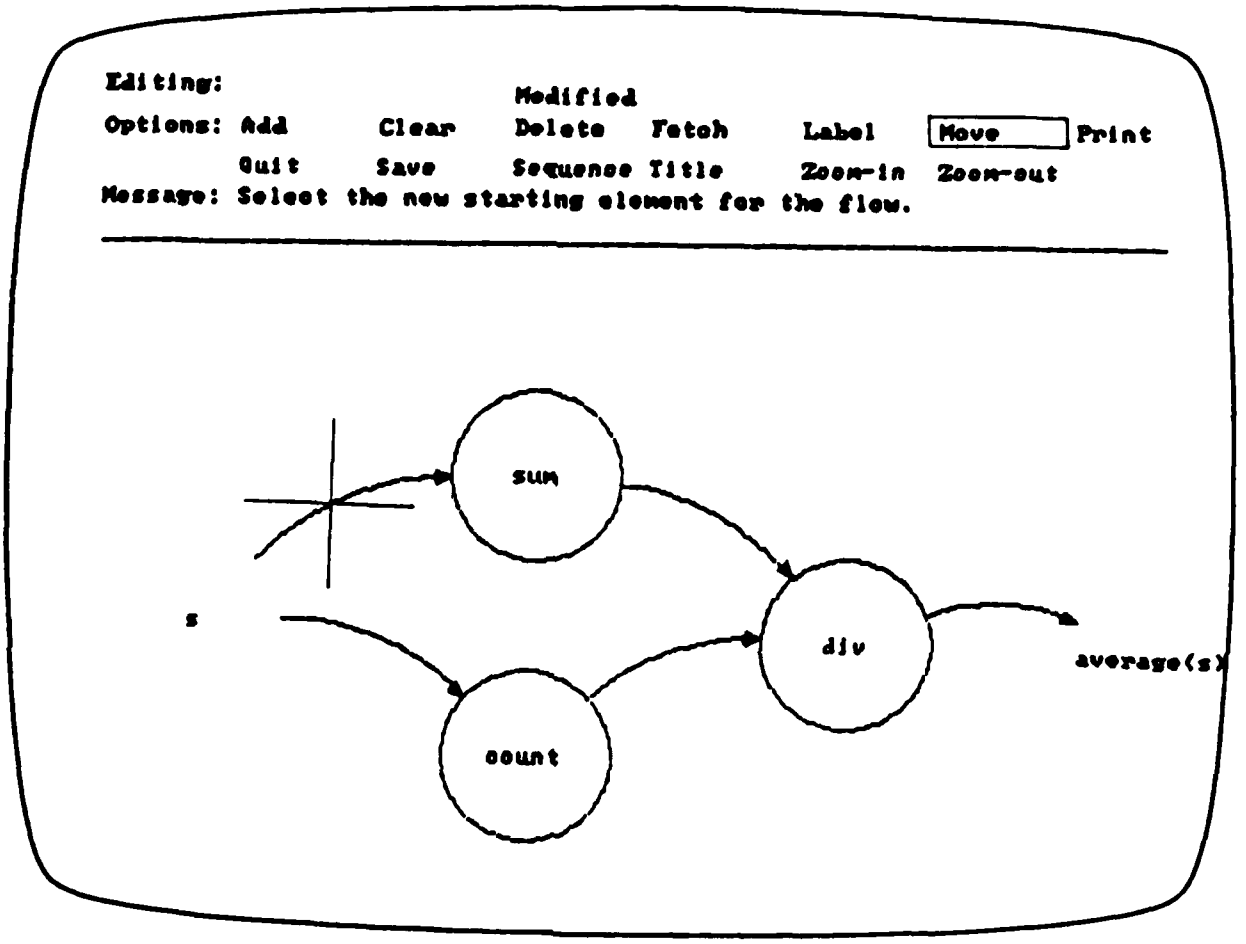
UNCLASSIFIED F/G 9/2 NL



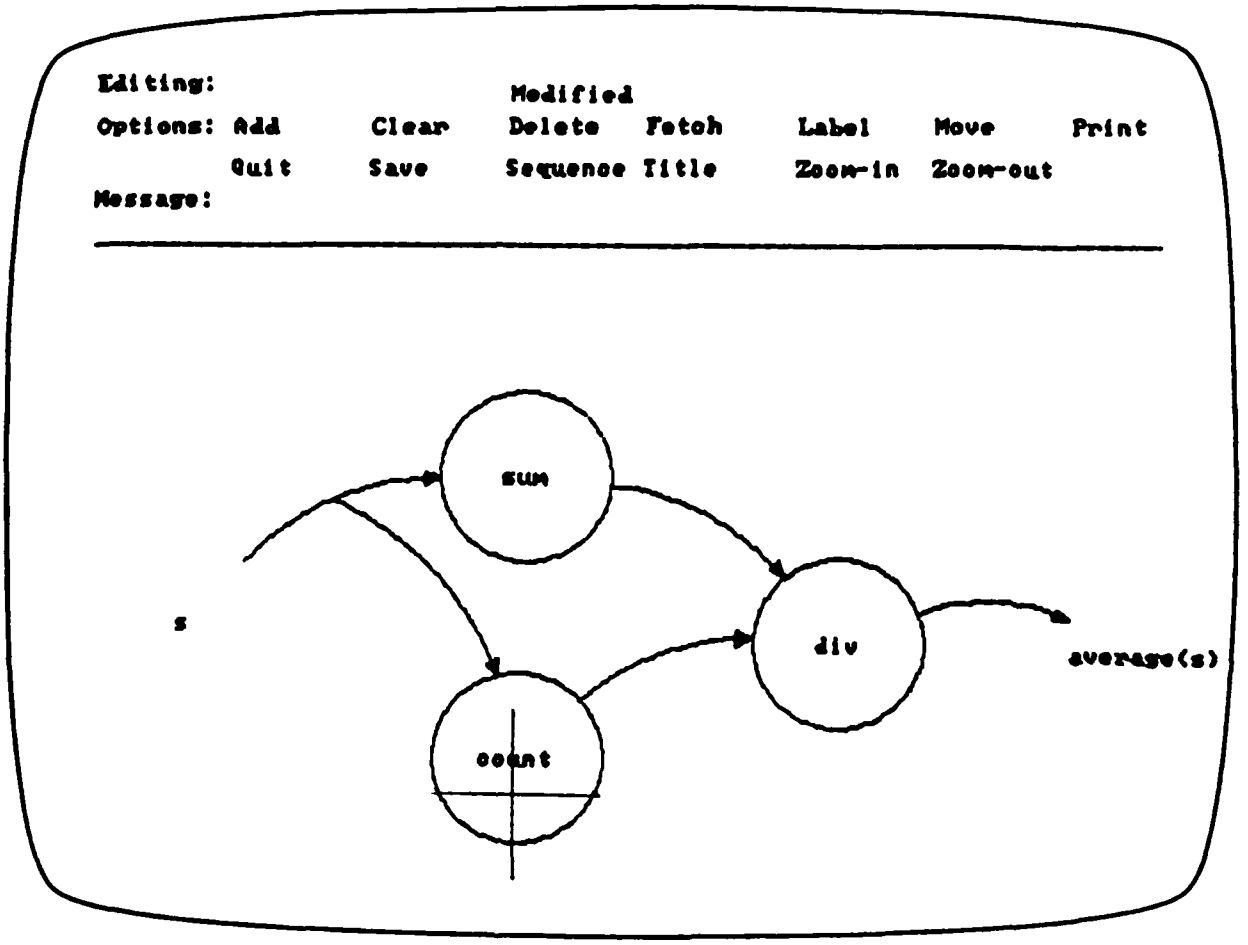


MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

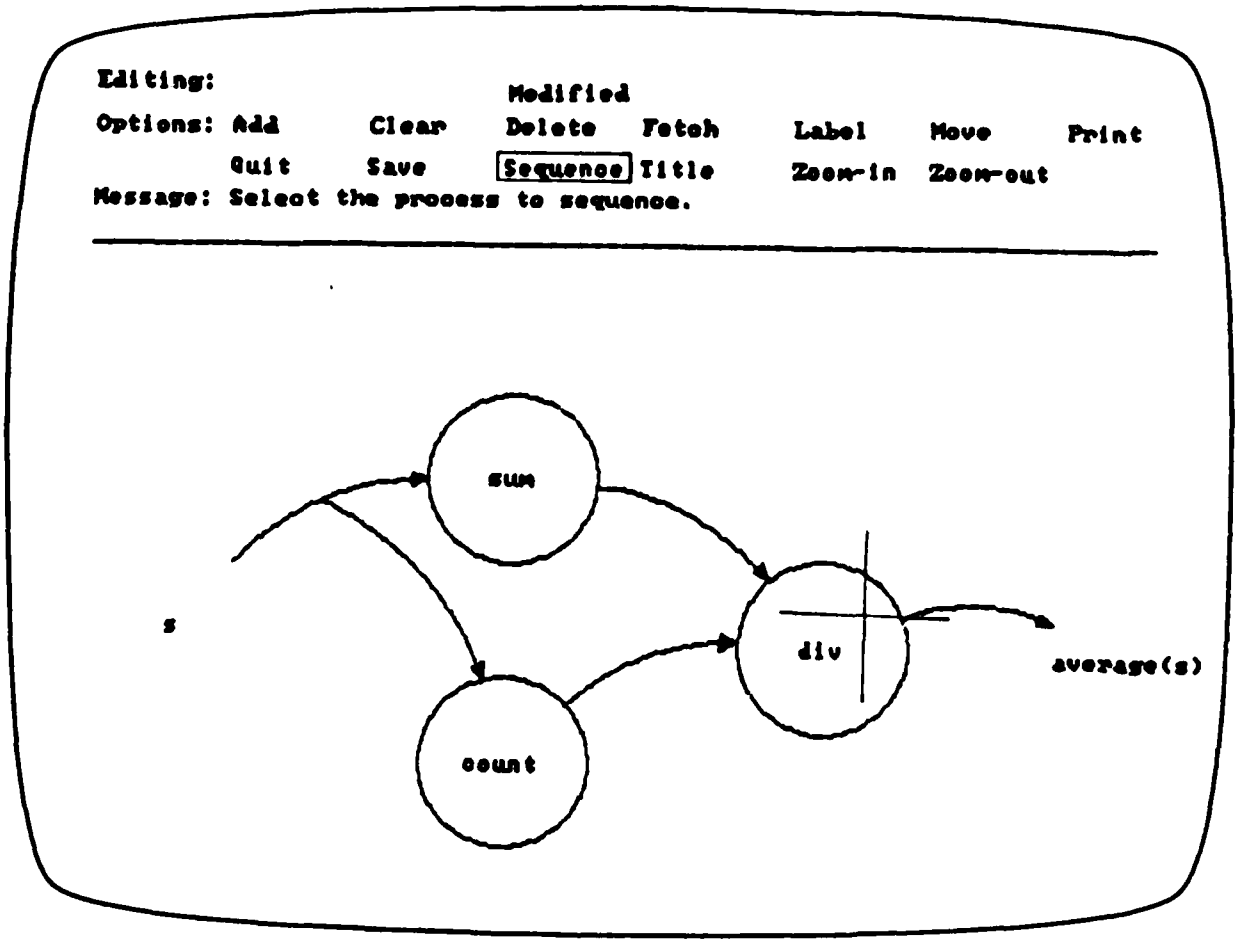
When the flow is selected, Van prompts you to select the new source element. Select the other flow originating at the interface "s."



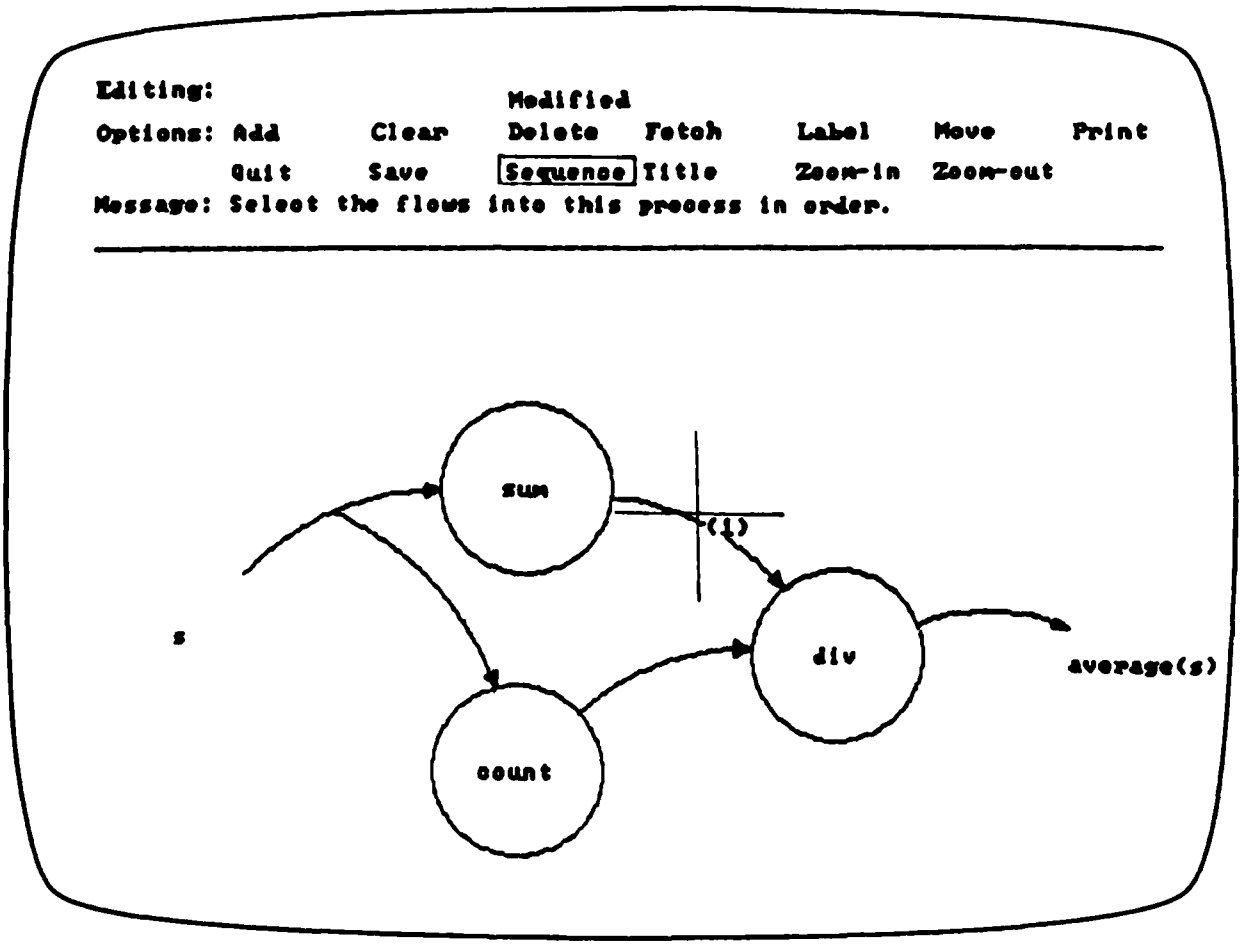
Van now prompts for the new destination element. Select the process "count". When this is selected, Van redraws the flow with its new orientation, completing the Move command.



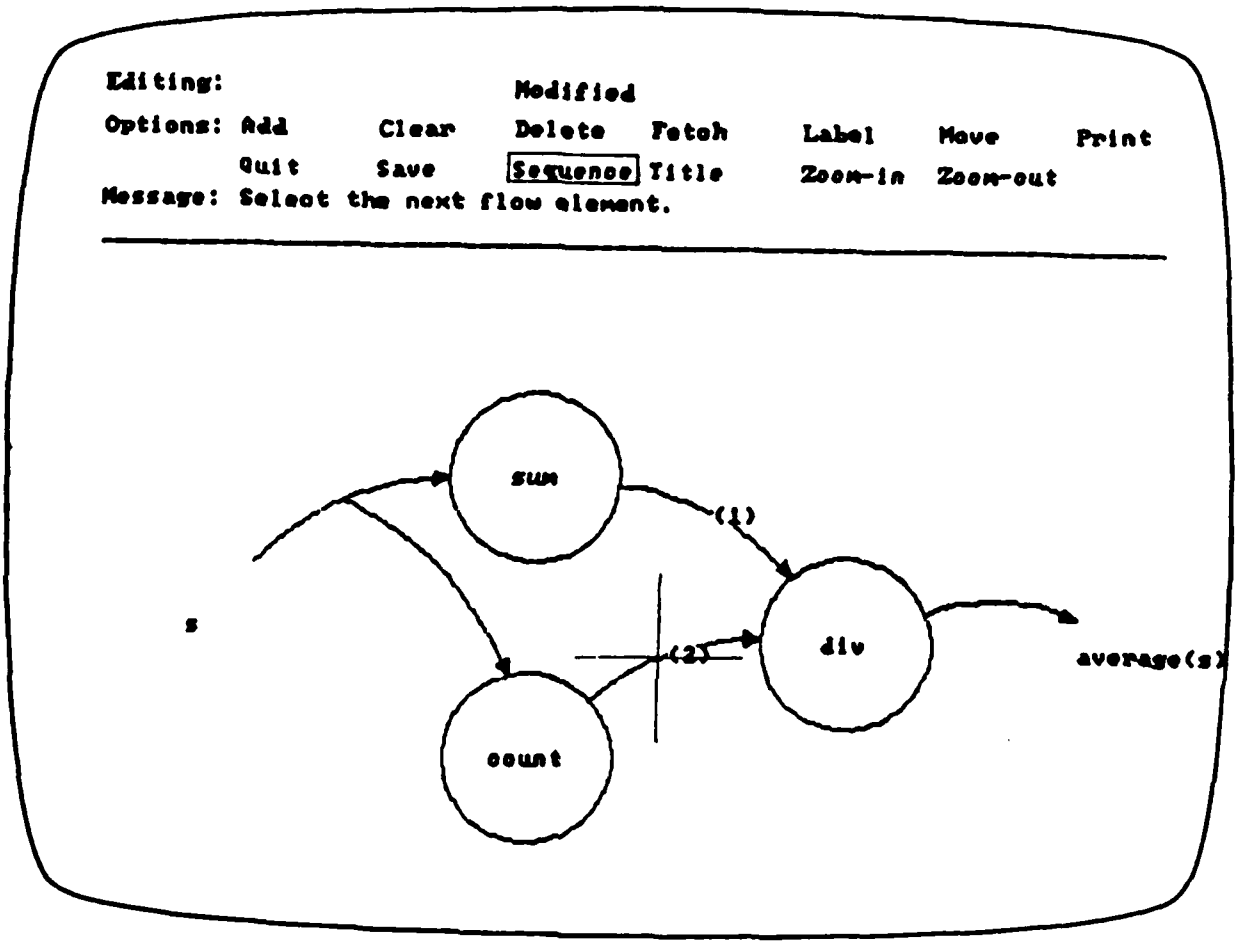
An additional cleanup item is to resolve ambiguities in the diagram. One such ambiguity is whether the diagram represents sum(s) divided by count(s) or count(s) divided by sum(s). Of course, the former is what we want and the ambiguity can be clarified by sequencing the flows into the process "div." When you select the "Sequence" command, Van displays the prompt, "Select the process to sequence." Select the process "div."



After you select the process "div", Van prompts you to select the flows into the process "div" in the order that you desire. You first want to select the flow from the process "sum" to the process "div."

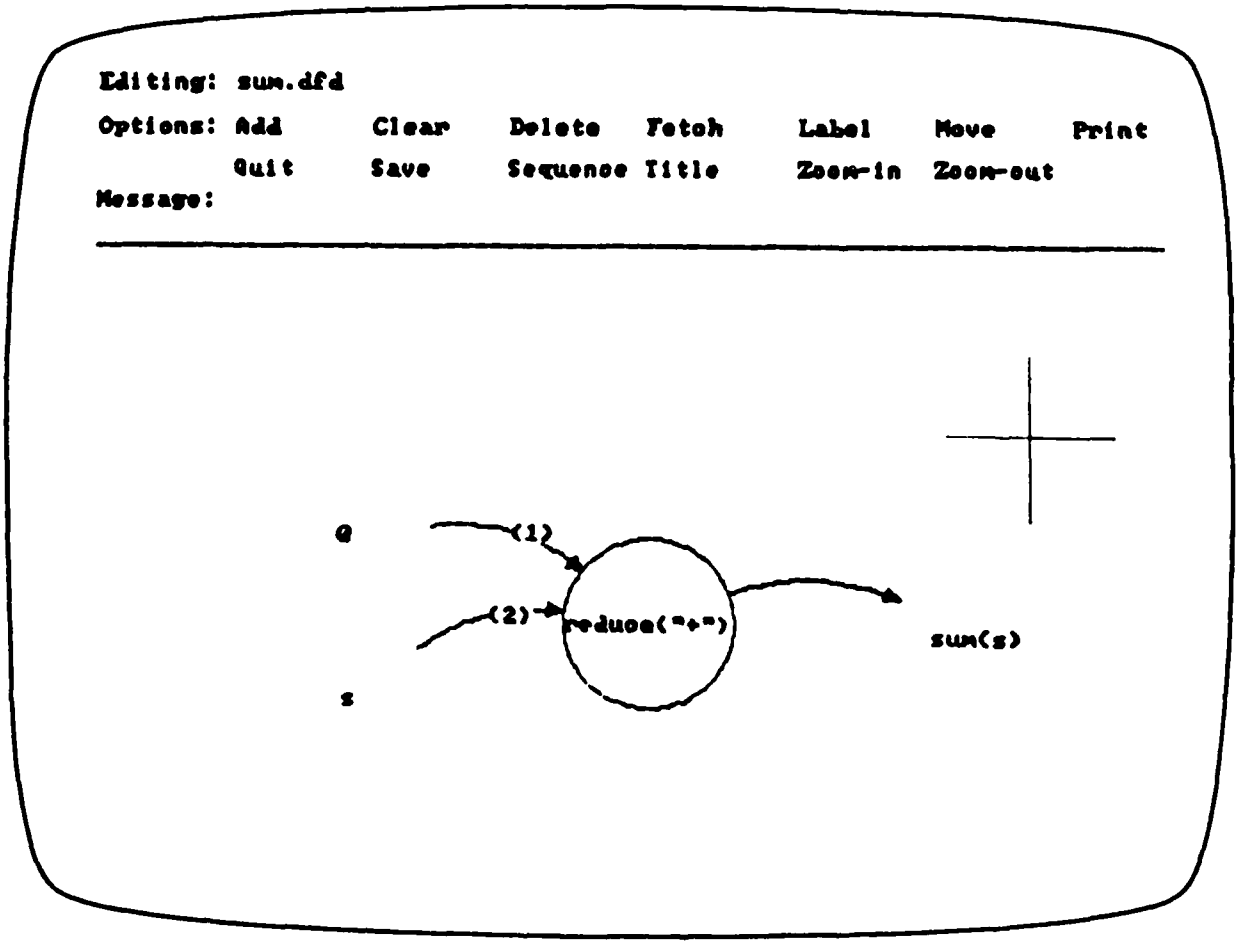


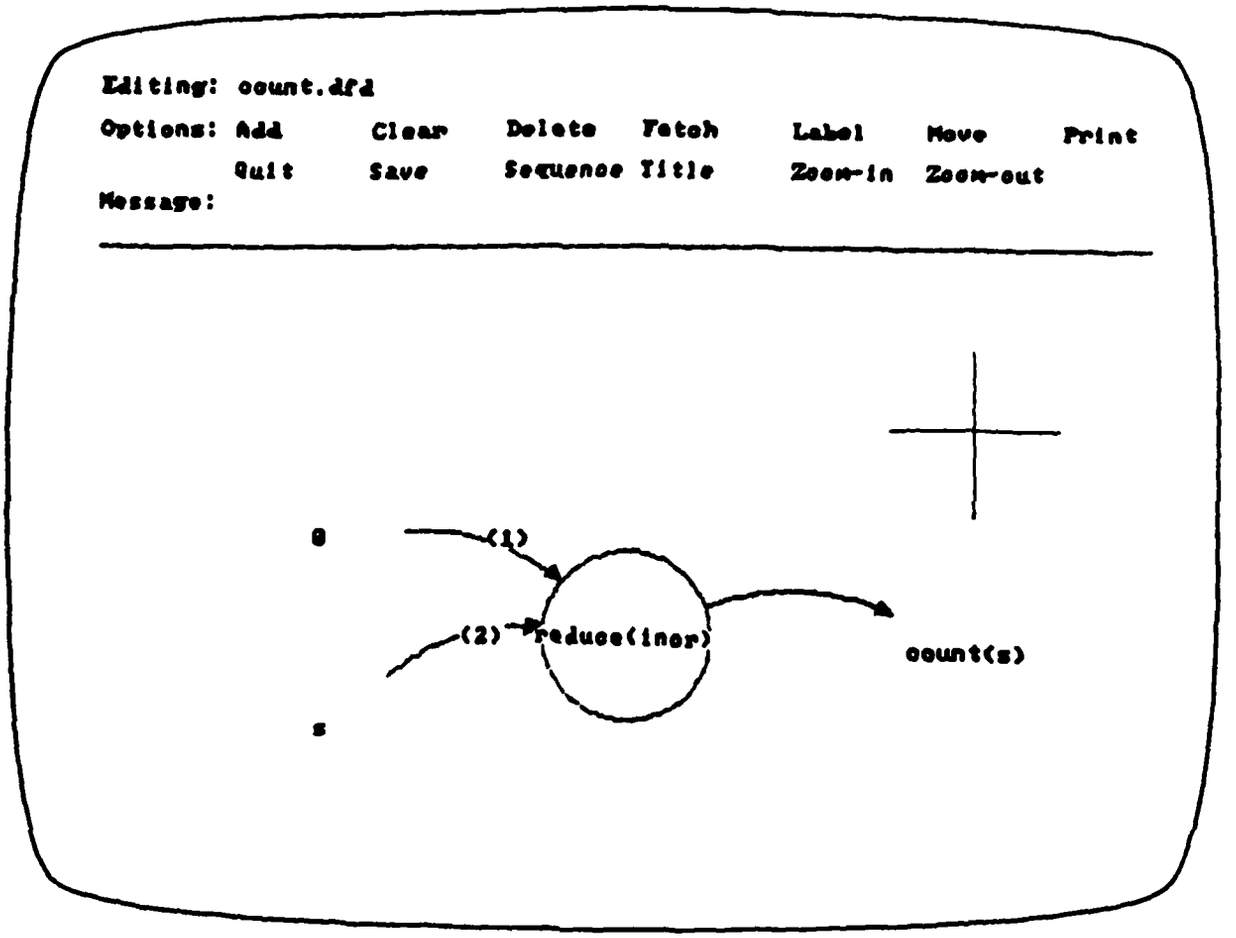
Then you want to select the flow from the process "count" to the process "div."



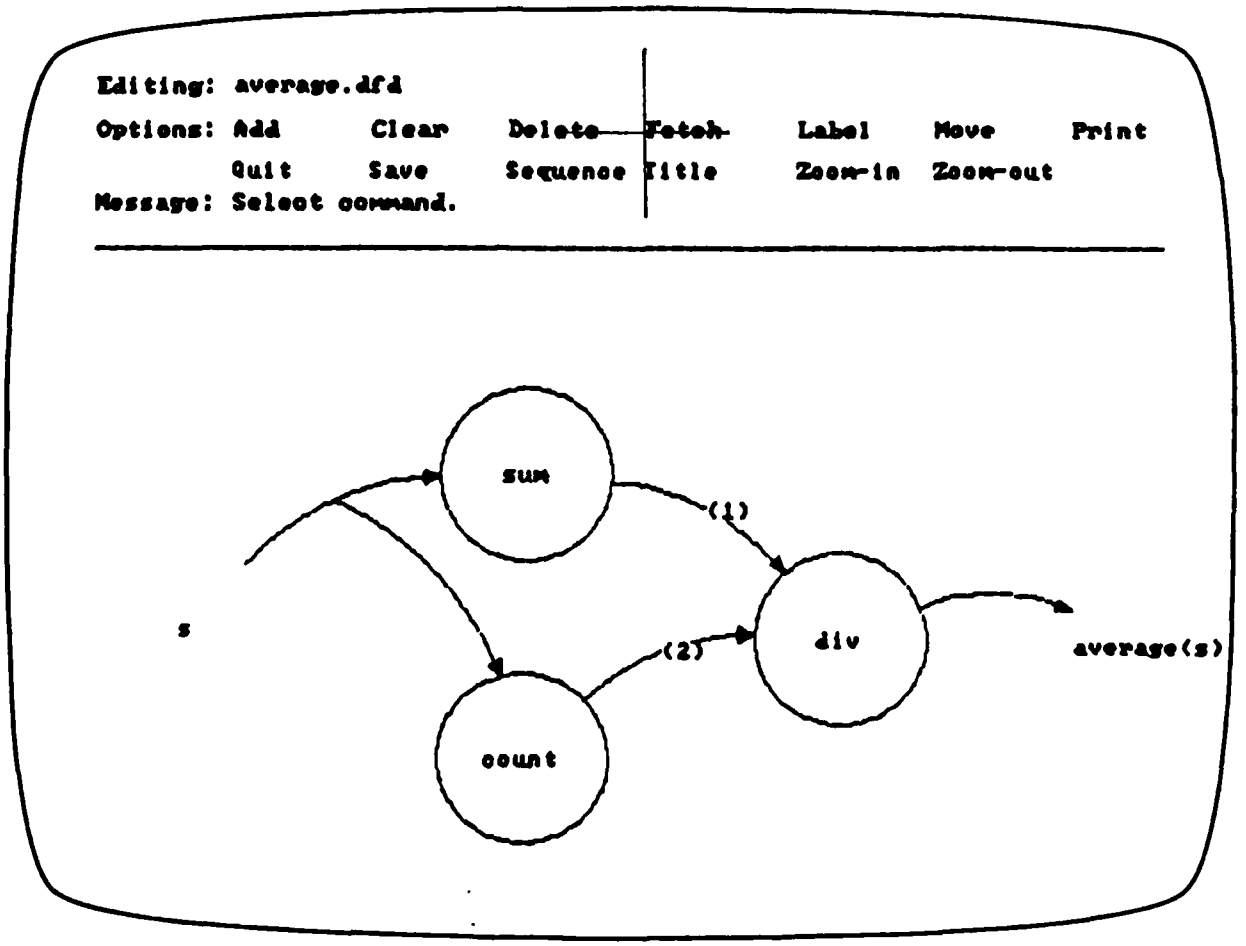
Notice that as you select the flows, Van displays the sequence number on the flow so that you can see the order you have specified. Now that you have sequenced both flows, cancel the command by moving the cursor into the menu area and pressing any button on the mouse.

Now that you have completed the data flow diagram for "average(s)", you can use the "Clear" command to start on the other two diagrams. The "Clear" command will remind you to save the diagram before Van clears the screen. The diagrams for "sum(s)" and "count(s)" are created in the same manner as discussed above.

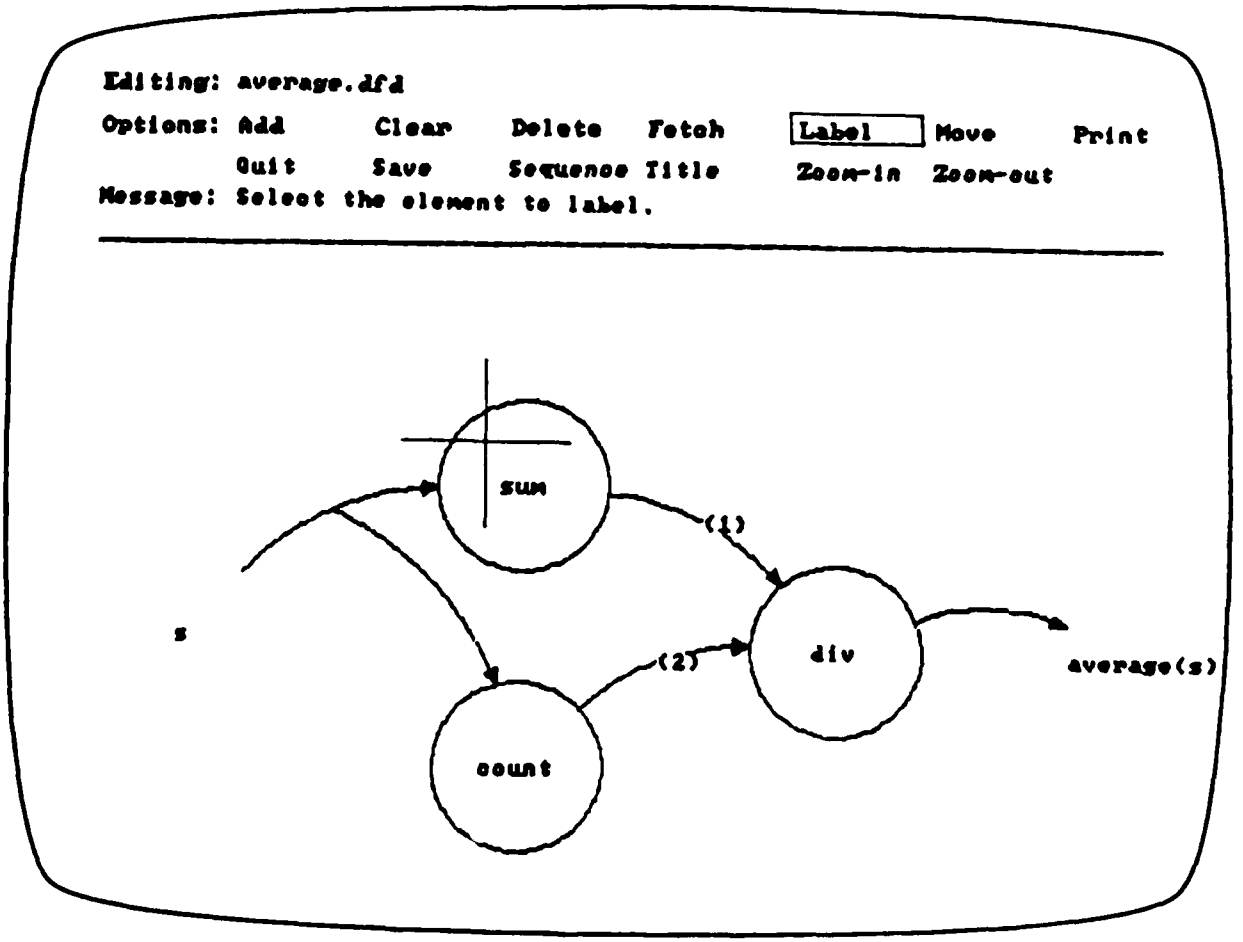



Editing: count.dfd
Options: Add Clear Delete Fetch Label Move Print
Quit Save Sequence Title Zoom-in Zoom-out
Message:
0
(1)
(2)
reduce(incr)
count(s)
s

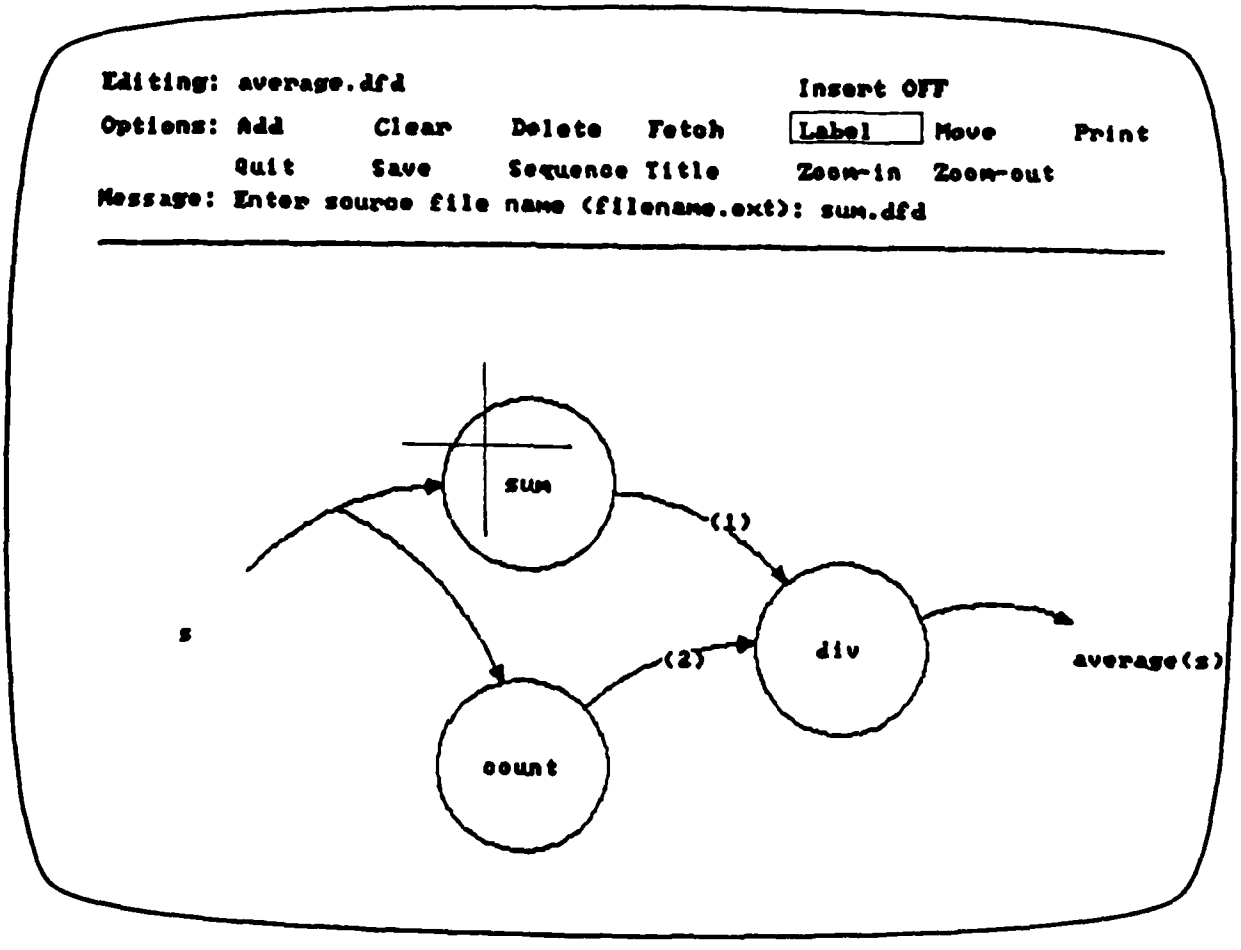
Since you now know the file names for the diagrams of "sum(s)" and "count(s)", you want to specify them as source file names in the diagram for "average(s)." First, you need to fetch the diagram for "average(s)". When you select the "Fetch" command, Van reminds you to save the current diagram first if you have not already done so. When the current diagram has been saved, Van prompts for the name of the file to retrieve. Once you specify the file name, Van clears the screen and displays the diagram for "average(s)."



Now select the "Label" command. Van prompts you to select the element to label. Select the process "sum."

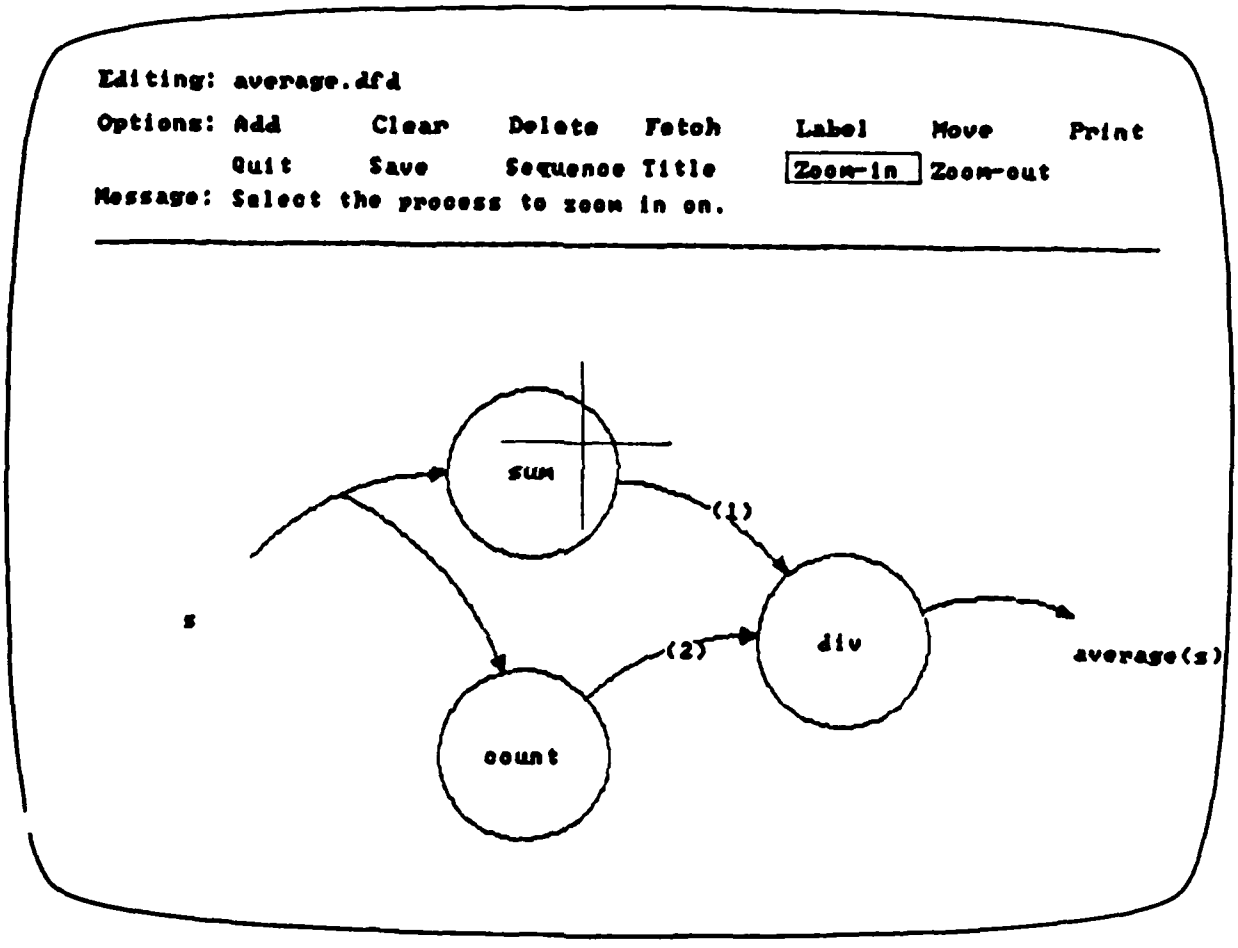


Van then displays, "Enter label: sum." Since you do not want to change the name of the process, respond by pressing Return. Van will then display the prompt, "Enter source file name (filename.ext):" for you to enter the source file name for the diagram of "sum(s)."

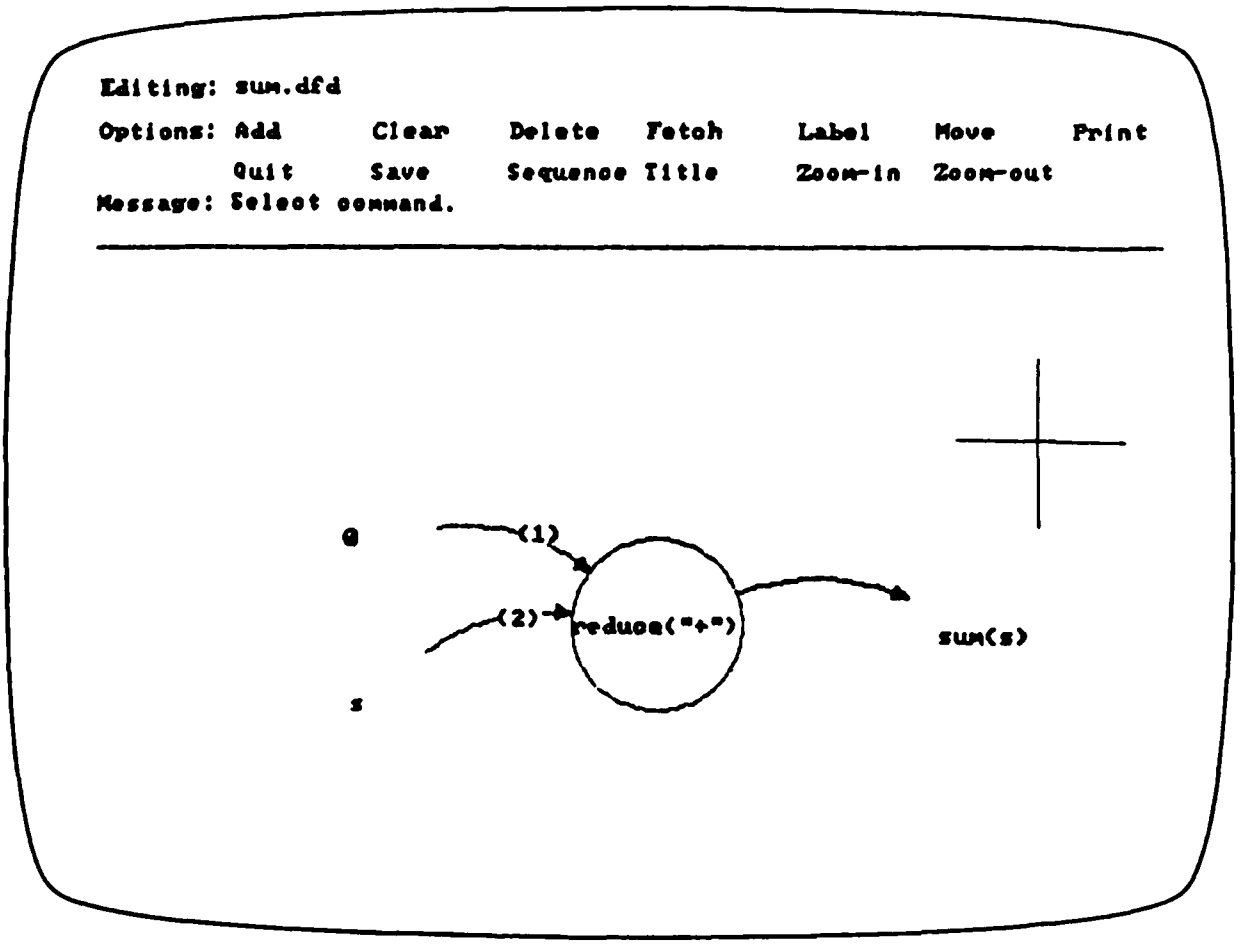


You can store the source file name for the diagram of "count(s)" in a similar manner.

Now that both source file names have been specified, you can travel from one diagram to another using the "Zoom-in" and "Zoom-out" commands. For instance, if Van is currently displaying the diagram for "average(s)", and you want to view or edit the diagram for "sum(s)", select the "Zoom-in" command. Van then prompts, "Select the process to zoom in on."



If you select the process "sum", for example, Van will display the diagram that defines "sum(s)."



You can print any of the three diagrams, with or without the sequence numbers on flows, with the "Print" command and save the diagrams for future use by selecting the "Save" command. Finally, you exit Van by selecting the "Quit" command which will return you to the operating system.

5. EXECUTABLE DATA FLOW DIAGRAMS

In this section we describe how data flow diagrams created using Van can be compiled into executable code. First we discuss the problems of interpreting ambiguous data flow diagrams and the solutions we have devised. Then we describe the procedures for compiling and executing data flow programs.

Ambiguitites in Data Flow Diagrams

To compile data flow diagrams into executable code, the diagrams must represent well-defined functions. Unfortunately, data flow diagrams often contain ambiguitites that would prevent their automatic translation.

One source of ambiguity is the order of inputs to a process. For example, the diagram in Figure 5-1 could mean "a-b" or "b-a". The data flow diagram compiler, not knowing the desired order of inputs to the process, will arbitrarily order the arguments for the subtraction. Hence, any time the order of arguments to a function is important, you should use Van's **Sequence** command to specify the required order.

A second source of ambiguity is that data flow diagrams can describe processes whose results depend on the timing of input and the speed of subprocess execution. This can occur when two or more processes share a common data store as shown in Figure 5-2. In this example, both Process_A and Process_B can read and update the shared store at any time, and can easily produce different results with the same input data if process timing changes. We solve this problem by not allowing data stores to be shared. Fortunately, this does not restrict the specification of programs that implement well-defined functions.

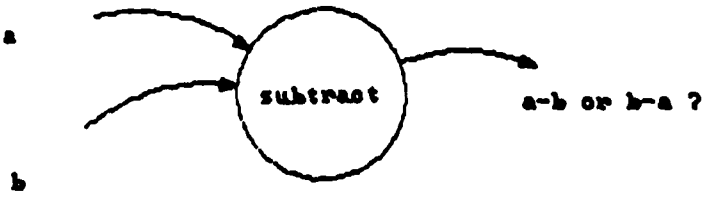


Figure 5-1. Ambiguity in the Order of Arguments

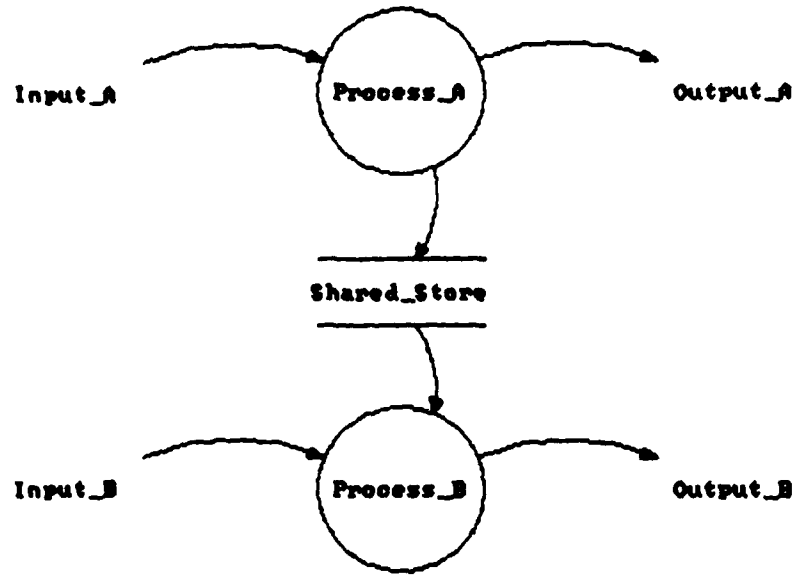


Figure 5-2. Ambiguous Process with Shared Data Store

Representing conditional processes in a data flow diagram is another source of ambiguity. The usual approach is to show multiple output flows to indicate conditional data paths, zero or more of which may be taken based on unstated criteria. Even when the branching conditions are specified, the compiler can run into difficulty trying to merge conditional flows back together. The convention we use is illustrated in Figure 5-3. It consists of a process labelled "if" which has three ordered inputs: the data flow for the condition (a true or false value), the data flow for the result if the condition is true, and the data flow for the result if the condition is false. Thus, the value on the output flow of the "if" process depends on whether the condition is true or false.

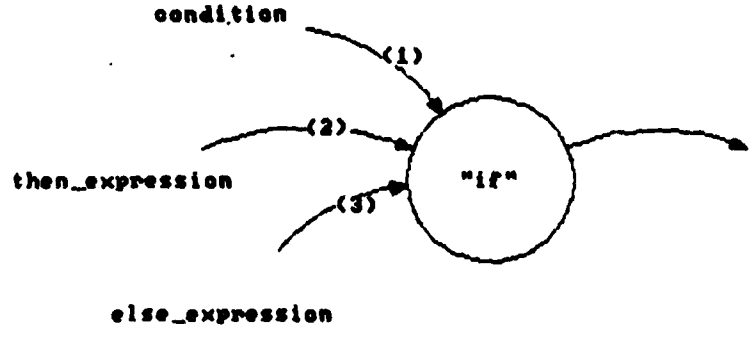


Figure 5-3. Representation of Conditional Data Flows

The current version of the data flow diagram compiler also requires adherence to the following conventions:

- Each data flow diagram should have only one output interface. This interface should be labelled with the function name and its list of formal parameters. (The title on a data flow diagram is ignored by the compiler and, hence, cannot be used to identify the function being defined.)

- All data flow diagram text should correspond to Ernest where clauses. That is, data flow diagram text should begin with the keyword **where** followed by an Ernest definition.

- Built-in operators, which are listed below, should appear in double quotes when used as process names.

"&"	"*"	"+"	"-"	"/"
"/="	":" (cons)	"<"	"<="	"="
">"	">="	"^" (concat)		
"if"	"mod"	"|" (or)	"~" (not)	

Compiling and Executing Data Flow Diagrams

The first step in compiling a data flow diagram is to derive an abstract syntax tree for the expression represented by the diagram. This is accomplished by a program called "Vincent." A sample command, which will create the abstract syntax tree for the function "average" is:

```
A> vincent average.dfd average.ast
```

The second step is to convert the abstract syntax tree into the low-level code understood by the run-time interpreter. This is accomplished by the second pass of the Ernest compiler, Oliver2, with the command:

```
A> oliver2 average.ast average.hbc
```

Since each data flow diagram specifies only a single function, this process must be repeated for each ".dfd" file.

The top-level diagram of a program must be handled slightly differently, since it represents an expression rather than a function definition. The translation process is exactly the same. However, the low-level code for the main program must be the last segment of code loaded by the run-time interpreter. To ensure that this code appears at the end of the file, we do not create a ".hbc" file for it. Instead, we use the file extension ".top", as in:

```
A> vincent mainprog.dfd mainprog.ast

A> oliver2 mainprog.ast mainprog.top
```

All of the ".hbc" files and the ".top" file must then be collected together into a single file for execution. The following command combines all the ".hbc" files in the current directory and the ".top" file into a single file called "mainprog":

```
A> copy *.hbc+mainprog.top mainprog
```

We can now run the program by issuing the command:

```
A> sherbert mainprog
```

VAN QUICK REFERENCE GUIDE

Add -- Add a data flow element to the current diagram.

- Addflow -- Add a data flow.
- Addiface -- Add an interface.
- Addproc -- Add a process.
- Addstore -- Add a data store.
- Addtext -- Add data flow diagram text.
- End -- End the Add command.

Clear -- Clear the screen.

Delete -- Delete a data flow element from the current diagram.

Fetch -- Retrieve a saved diagram.

Label -- Add or Edit a data flow element label.

Move -- Move a data flow element.

Print -- Print the current data flow diagram.

Quit -- Quit the editing session. Exit Van.

Save -- Save the current data flow diagram.

Sequence -- Sequence the flows into a process.

Title -- Add or Edit the data flow diagram title.

Zoom-in -- Move down one level in the diagram.

Zoom-out -- Move up one level in the diagram.

Annex C. APPLICATION PROGRAMS

HUMAN FACTORS DESIGN EVALUATION TOOL

Reginald N. Meeson, Jr.
T. Patrick Gorman

Computer Technology Associates, Inc.
7501 Forbes Blvd., Suite 201
Lanham, MD 20706
301-464-5300

Abstract

A sizable application program (1700 lines) written entirely in the functional programming language Ernest is described. The application is an interactive tool for analyzing the effects of automation on personnel at a NASA ground control center, and is based on a human factors model of job activities. The functional organization of the program is discussed and several examples of function definitions and data structures are presented. Ernest's function-forming operations allowed us to abstract and partition the problem in a logical and natural way. The techniques of functional programming which have been illustrated in much smaller programs worked equally well in this larger exercise.

Introduction

This paper describes an interactive tool designed to analyze the effects of automating job activities on personnel and operations at a NASA ground control center. Changes in job activities due to automation can have both positive and negative effects on personnel and overall system performance. Hence, analysis of such effects can be valuable to help guide system evolution and to safeguard against having to correct for changes that impair performance.

The analysis is based on a model of human factors attributes associated with personnel functions and tasks that support ground control system operations [1]. The model partitions systems into the hierarchy of components outlined in Figure 1. Each component in this tree structure has a set of attributes. At the leaves of the tree, the attributes are directly measurable, quantitative characteristics of a human-computer interface. At other nodes, the attributes are derived from the attributes at lower levels and are usually more qualitative in nature. At the root node, therefore, the attributes represent an overall qualitative assessment of the system's demand on personnel and its effects on personnel performance. The attributes at each level in the hierarchy and their dependencies are shown in Figure 2.

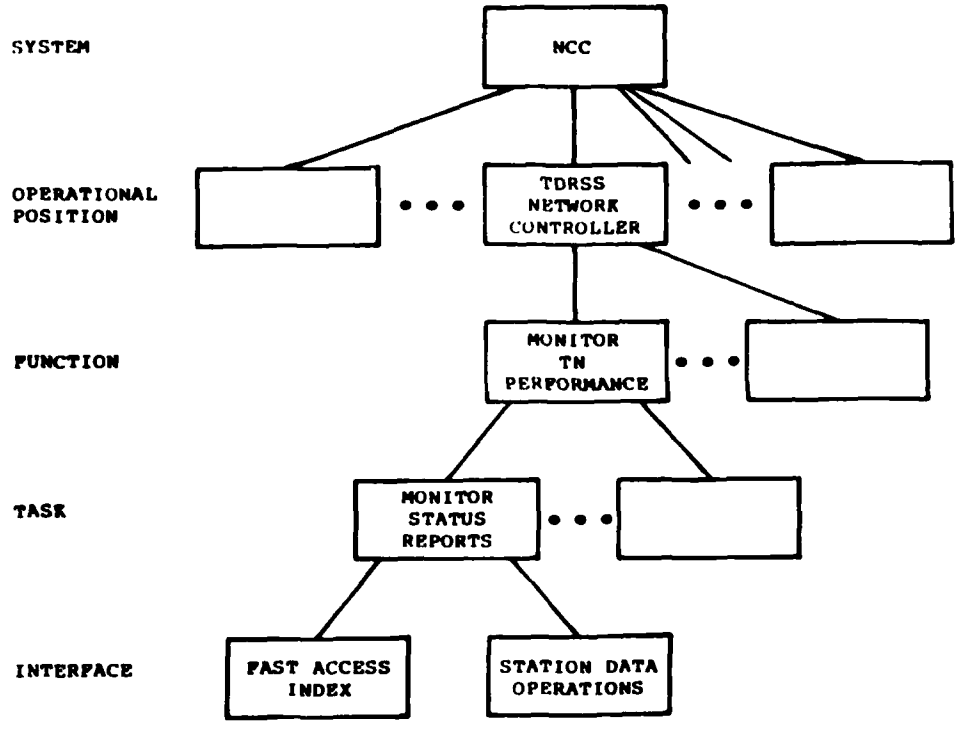


Figure 1. Model Personnel Functions and Tasks

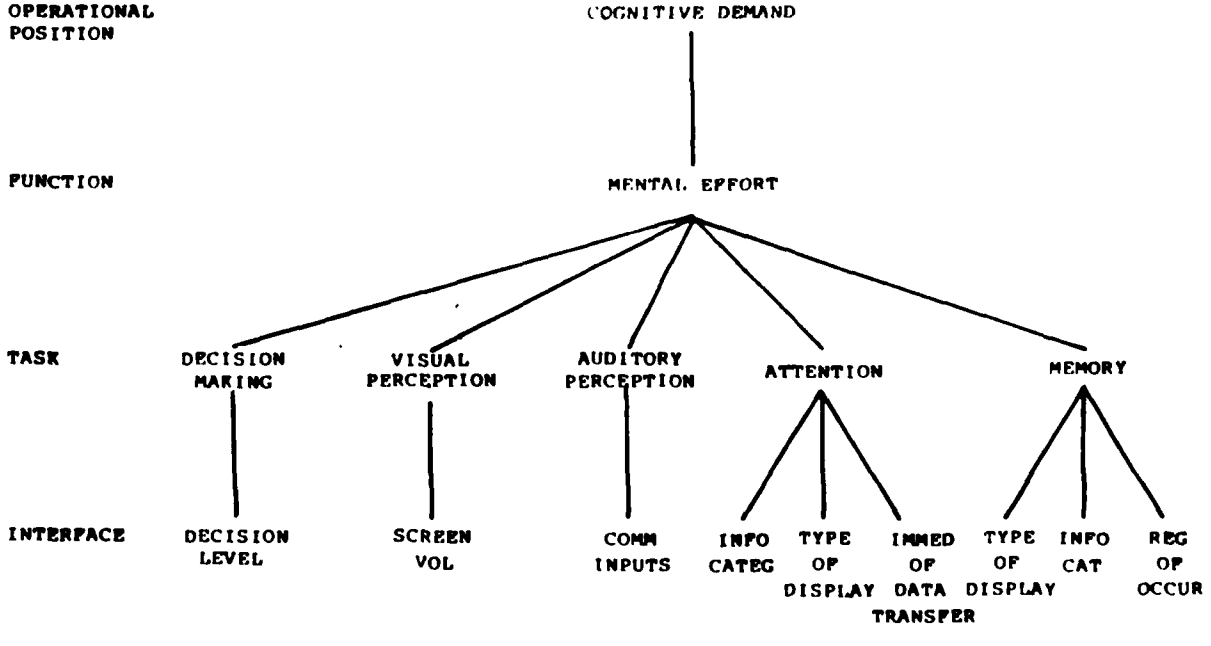


Figure 2. Attributes Associated with Model Components

The Design Evaluation Tool allows a user to move around within this model and display information about the hierarchical structure, the values of attributes, and the contributions lower-level nodes make to attributes at higher levels. In addition, a user may experiment with changing attribute values at leaf nodes to investigate the effects of changes in work activities on personnel and system performance. Examples of the two principal types of displays produced by the tool are shown in Figures 3 and 4.

The Program

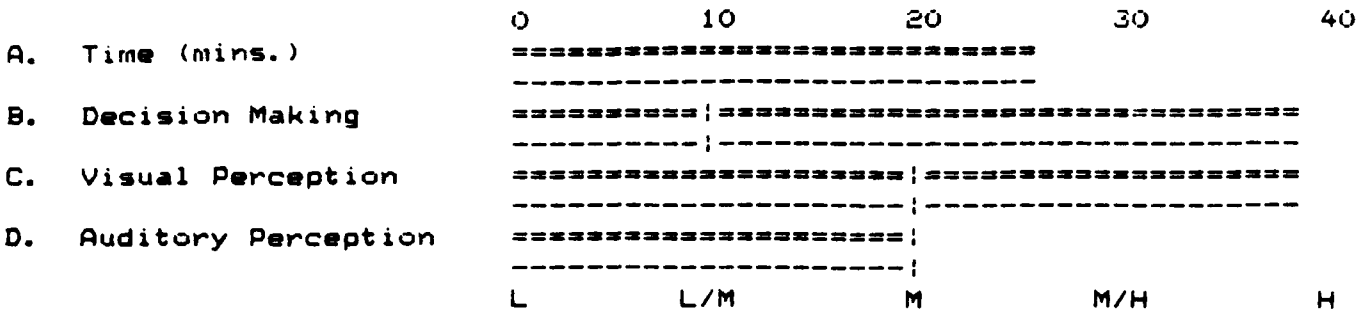
The Design Evaluation Tool is implemented entirely in the functional programming language Ernest [2]. The program consists of approximately 1000 lines of function definitions, plus approximately 700 lines of data structure definitions which form the model of human factors attributes.

The top-level organization of the program has the traditional input-process-output structure shown in Figure 5. The first function, "make_codes", filters the user's keyboard entries and passes a command code to the next stage for each user entry. The process part, which contains the function "next_state", implements a finite-state machine that produces a new state in response to each input command code. On the output side, the function "show_state" displays the current state on the user's terminal screen.

```
Function:  Monitor TDRSS Network Performance

To exit, enter ^C.

To proceed, select an action from the list below:

   1.  Move to Task:  Monitor Status Reports
   2.  Move to Task:  Maintain Position Log
   3.  Move to Task:  Monitor/Control GCMS
   P.  Display the Profile for this node.
   R.  Return to the previous screen.
```

Figure 3. Example Menu Node Display

Profile for Task: Monitor Status Reports



Demand: ! is recommended; = is baseline; - is trial modification.

To exit, enter ^C. To return to previous screen, enter R.

To proceed, select an attribute for analysis or modification.

Figure 4. Example Profile Node Display

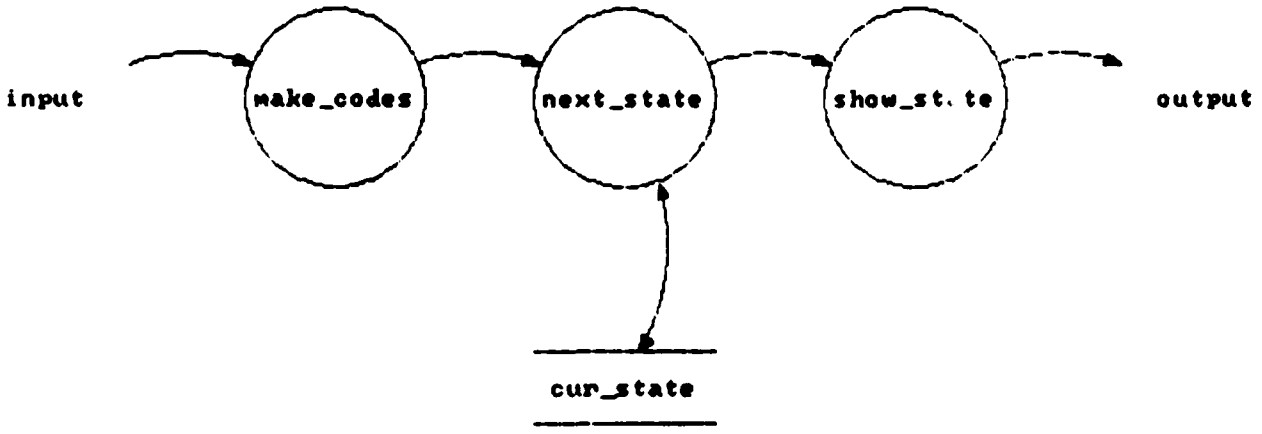


Figure 5. Top-Level Program Organization

Data Structures. Before we say more about these functions we must describe some of the data they operate on. For each of the nodes in the model there is a menu node in the program database. Each menu node contains a tag field indicating that it is a menu, its name, a list of its offspring, a list of its attributes, and a function that computes its attribute values when applied to its offspring. For example, the menu node for the task "Monitor Status Reports" is:

```
MSR_menu  =  < 'M', 'Task: Monitor Status Reports',
               offspring, attributes, task_attribs >

where  offspring = < <'1',FAI_menu>,   -- Fast Access Index
                     <'2',SDO_menu>,   -- Station Data Ops.
                     <'P',task_profile>,
                     <'?',nono_msg> >

and  attributes = task_attribs(offspring)
```

Since the attribute function is the same for all tasks, the function "task_attribs" is defined globally. In addition to subordinate menu nodes, the offspring include a profile node and a text node.

Profile nodes are not described in the model but are required in our finite-state machine paradigm to allow profiles of attributes to be displayed. The definition of the task profile node is:

```
task_profile  =  < 'P', 'Task Profile:', offspring >

where  offspring = < <'A',task_A_detail>,
                     <'B',task_B_detail>,
                     <'C',task_C_detail>,
                     <'?',nono_msg> >
```

Profile nodes have offspring which are detail nodes. Detail nodes allow us to display the contributions made by each subordinate node to a selected attribute.

The "state" of the finite-state machine part of our program is a list of the nodes that lead from the current node back up to the root of the tree. For example, if the current node is the profile node for the model function "Monitor TN Performance" then the state would be represented by:

```
cur_state  =  < func_profile,    -- function profile node
              MTNP_menu,         -- Monitor TN Performance
              TNC_menu,          -- TDRSS Network Controller
              NCC_menu >         -- Network Control Center
```

Function Definitions. Space does not allow us to show every detail of the program, so we will attempt to describe only a few key functions that illustrate program construction using function-forming operations. All of these examples were taken directly from the program and are compilable definitions.

The heart of this program is the finite-state machine shown in the center of Figure 5. Given the current state and an input command code, the function "next_state" produces a new state value. That is,

```
new_state  =  next_state( cur_state, cmd_code )
```

We use the "scan" operator to construct the finite-state machine. "Scan" is the function that produces the sequence of all its partial results as it consumes its input. In this program, the partial results are the individual state changes directed by each input command code. Hence,

```
state_stream  =  scan( next_state, init_state, cmd_stream )
```

is a stream of state values starting with the initial state and containing every intermediate transition. The initial state is a list containing only the root of the database tree structure.

The definition of "next_state" does not use function-forming operations, but it does show how the program state is manipulated and how functions may return large data structures as results.

```
next_state( cur_state, cmd_code )  =
     if  cmd_code = 'R'
        then   -- return to parent node
        if  parent_state = <>    -- cur_state = <root>
           then  cur_state
        else  parent_state
     else if  node_type = 'M' |    -- a menu node or
              node_type = 'P'      -- a profile node
        then   -- add the selected offspring to the state
        select_offspring(cur_node,cmd_code) : cur_state
     else if  node_type = 'T' |    -- a text node or
              node_type = 'D'      -- a detail node
        then   -- return to the parent node
        parent_state
     else if  node_type = 'U'    -- an update node
        then   -- produce an entirely new state
        update_state(cur_state,cmd_code)
     else   -- error, replace node with error message
        oops_msg : parent_state

where  cur_node : parent_state = cur_state

and  node_type = head(cur_node)

and  oops_msg = <'T', <'Oops! Unrecognized node in tree'> >
```

Selecting an offspring from the current node requires matching the command code with an offspring node's selection key. This function is defined by:

```
select_offspring( cur_node, cmd_code )  =  new_node

where  <key,new_node> = first(match,offspring)

and  first(pred,seq) = head( filter(pred,seq) )

and  match(<key,node>) =
          key = cmd_code |    -- key matches command code
          key = '?'           -- or the last element

and  <type,offspring,name,attributes,attr_func> = cur_node
```

The first node that matches is at the head of the list of all the nodes that match, which we produce using the "filter" function-forming operation.

Since Ernest has no assignment statements, we cannot update attribute values in place the way we can in conventional languages. Instead, we must produce a new state with appropriately reconstructed nodes. (Assignment statements over-simplify the drastic nature of global state transformations. In this program, for example, the effects of attribute changes ripple all the way up the tree structure. Hence, a simple assignment would leave the program database in an inconsistent state.) Updating a menu node consists of replacing its offspring and recomputing its attribute values:

```
update_menu( parent_node, new_child )  =
     < 'M', new_offspring, name, new_attribs, attrib_func >

where  <type,offspring,name,attributes,attrib_func>
            = parent_node

and  new_offspring = replace(new_child,offspring)

and  new_attribs = attrib_func(new_offspring)
```

The function "replace" substitutes a new child in the list of choices that make up a parent node's offspring. It is defined recursively by:

```
replace( new_child, choices )  =
     if  choices = <>  then  <>
     else if  new_name = first_name
        then   -- insert the new child here
        <key,new_child> : rem_choices
     else   -- keep the first child and replace the rest
        first_child : replace(new_child,rem_choices)
```

```
where  <new_type,new_offspr,new_name,new_attrs,new_func>
           = new_child

and  <key, <first_type,first_offspr,first_name,
             first_attrs,first_func> >  =  first_choice

and  first_choice : rem_choices = choices
```

At the bottom level of the tree, a new node with new trial attribute values is constructed by the function "update_trial." This produces the first new menu node to be replaced in the tree structure. All of this child's ancestors -- back to the root of the tree -- are in the program state. This leads to the following definition:

```
update state( cur_state, cmd_code ) =
     update_node : profile : new_ancestors

where  update_node : profile : first_menu : ancestors
         = cur_state

and  new_ancestors = reverse( reduce(build_state,
                                     <first_new_menu>,
                                     ancestors) )

and  first_new_menu = update_trial(first_menu,attr_code,
                                   cmd_code)

and  <type,attr_code> = update_node
```

The function "build_state" takes a list of updated children and the next parent node in the tree structure, forms a new parent node with the newest updated child substituted in its offspring, and adds the new parent node to the list of updated children.

```
build_state( updated_children, parent ) =
     new_parent : updated_children

where  new_parent = update_menu(parent,first_child)

and  first_child = head(updated_children)
```

Applying "build_state" successively to each ancestor in the state using the reduction operator constructs a new updated state -- but with the elements in reverse order. The function "reverse" rectifies this:

```
reverse = reduce( rev, <> )

where  rev(seq,elem) = elem : seq
```

The states produced by "next_state" are passed along to the output formatting function "show_state", which is defined by:

```
show_state( cur_state ) =
     screen( if node_type = 'M' then        -- it's a menu
                show_menu(cur_state)
             else if node_type = 'P' then   -- it's a profile
                show_profile(cur_state)
             else if node_type = 'D' then   -- it's a detail
                show_detail(cur_state)
             else if node_type = 'T' then   -- it's text
                show_text(first_node)
             else if node_type = 'U' then   -- it's an update
                show_update(cur_state)
             else show_text(error_msg) )
```

Each of the subordinate "show_..." functions produces a sequence of lines to be displayed, and "screen" centers these lines vertically to present a clean display.

Finally, the program output is generated by mapping "show_-state" across the sequence of states produced by the finite-state machine:

```
map( show_state, state_seq )
```

Conclusion

We have described a sizable application program (1700 lines) which was written entirely in the functional programming language Ernest. The success of this project was due to several factors. Ernest allowed us to abstract and partition the problem in a logical and natural way. Ernest's function-forming operations and the absence of side effects allowed us to build and test component functions independently and to combine them into larger components with ease. The same techniques that we used in developing smaller programs worked equally well in this larger example. We also found that program bugs were quite easy to isolate and repair, primarily because they were found early in testing small components.

References

1. Sheppard, S., E. Murphy, and L. Stewart, "A Methodology for Assessing the Human Factors Impacts of Increased Automation", Proc. Human Factors Soc. 29th Annual Meeting, October, 1985, pp. 556-560.

2. Meeson, R., Ernest Functional Programming User's Guide, Computer Technology Associates, 1986.

TYPE CHECKING IN ERNEST

Michael B. Dillencourt
Reginald N. Meeson, Jr.

Computer Technology Associates, Inc.
7501 Forbes Blvd., Suite 201
Lanham, MD 20706
301-464-5300

Abstract

A prototype of the type-checking algorithm used in the functional programming language Ernest is described. This prototype was implemented in Ernest as part of an exercise to evaluate the language. Since Ernest requires a minimum of type declarations, type checking requires deriving a function's type from its definition, as well as verifying that each application is consistent. The type checker supports polymorphic functions and derives the most general interpretation of each function's type.

Introduction

The goal of type checking is to ensure that functions and operators are applied only to arguments of the correct type. For example, it is valid to add two numbers, but it is not meaningful to add a boolean value to a string (or to another boolean value). In this paper, we discuss the approach to type checking in the functional programming language Ernest [1] and provide an overview of how our type checking algorithm works.

Most conventional programming languages support type checking by requiring programmers to explicitly declare the types of all variables, procedures, and functions. This makes type checking a simple process, since all the type checker has to do is verify that the use of each variable is consistent with its declaration.

The philosophy used in Ernest and in a number of other new programming languages is that the type of function arguments and returned values should be derived or inferred from the function's definition. Functions, therefore, do not require explicit type declarations. (This philosophy extends to procedures and variables in procedural languages.)

An important concept that arises in type systems is that of polymorphism. A polymorphic function is one that can be executed for arguments of different types, such as the following function to compute the minimum of two values:

```
define min(x,y) = if x <= y then x else y
```

This function is polymorphic because it can take two arguments of any type for which comparison is supported (e.g., integers or strings). Thus, both of the following calls are valid:

```
min('a','c')   -- returns the value of the smaller character
               -- (according to the ASCII collating sequence)

min(23,15)     -- returns the value of the smaller integer
```

Most conventional programming languages fail to support polymorphism, although Ada [2] provides a restricted form through "generic" objects and by "overloading" function and procedure definitions. Ada's approach, however, requires more, complicated declarations rather than fewer, simpler ones.

Another important concept in type systems is the notion of type equivalence. In Ernest, types are equivalent if they have the same structure; that is, if they are the same basic type or if they are built from equivalent components using the same construction operations. Other languages such as Ada have "stronger", more restrictive definitions of type equivalence which require more elaborate declarations. The weaker form of structural equivalence is required to allow us to derive anonymous types for functions.

Types in Ernest consist of type expressions built out of the following types and type construction operations:

- **Basic types**, which are integers, reals, booleans, characters, and strings.

- **Mappings**, which are denoted by the symbol "->" and represent the types of functions in terms of the domain and range types. Thus the logical negation function "~" has the type

  ```
  boolean -> boolean
  ```

  A slightly more complicated example is the function "+", which takes two numeric arguments and returns their sum. Its type is

  ```
  (number x number) -> number
  ```

  The Curried interpretation [2] of this function, which is the interpretation we use, has the type

  ```
  number -> (number -> number)
  ```

- **Sequences** of objects of a given type. For example, strings are sequences of characters.

  string = sequence(character)

- **Type variables**, which are used to describe the types of polymorphic functions. These are denoted by Greek letters (written "alpha", "beta", etc.). For example, the function "min", defined above, is of type

  alpha -> (alpha -> alpha)

  where "alpha" can be any ordered type. Similarly, the function "head", which returns the first element in a sequence, is of type

  sequence(beta) -> beta

- **Error**, which is the type of any expression that contains a type error. For example, the expression

  3 + 'blind mice'

  is of type "error", since a number cannot be added to a string.

Substitution, Instances, and Unification

The algorithm for type checking in Ernest is based on the concepts of substitution, instances, and unification. More complete treatments of these concepts can be found in the literature [3,4]. A **substitution** is a consistent assignment of type expressions to variables within a type expression, yielding an **instance** of the expression. Thus, "string -> string" is an instance of "alpha -> alpha", but "string -> integer" is not. The first expression is an instance because "string" was substituted for both (all) occurrences of "alpha." The second expression fails to be an instance because the substitution is not consistent. ("String" is substituted for the first occurrence of "alpha", and "integer" is substituted for the second.)

Unification of two given type expressions consists of finding a set of substitutions that yield a common instance of both given expressions. For example, consider the following two expressions:

string -> alpha

beta -> (string -> string)

These expressions can be unified by replacing "alpha" by "string -> string" in the first expression and substituting "string" for "beta" in the second, yielding the common instance

string -> (string -> string)

The following example shows two expressions that cannot be unified because no substitutions of alpha and beta will produce a common instance.

string -> (alpha -> string)

integer -> (integer -> beta)

The Function "Unify"

Unification is easily defined as an Ernest function. The function "unify" takes three arguments: a list of substitutions "S", and two type expressions "m" and "n" to be unified. "S" represents all the substitutions that have already been made. Each such substitution consists of a variable and a type expression to be substituted for that variable. "Unify" returns a list containing two items: an augmented substitution list and a type expression. The augmented substitution list contains everything in "S" together with all new substitutions made in unifying "m" and "n." The type expression is the common instance of "m" and "n" if one is found; otherwise it is the type "error."

We are almost ready to describe how "unify" works, but we need to explain one subtlety first. We have to make sure that the substitutions we make when we unify "m" and "n" are consistent with substitutions that have already been made (i.e., with those in "S"). This can be achieved by doing the unification on the respective representatives "e" and "f" of "m" and "n" from the substitution list "S." Thus "e" is the type expression corresponding to "m" in "S" if there is such a type expression; otherwise it is "m" itself.

Unification works as follows. Representatives "e" and "f" can be unified only if they are the same kind of type expression (e.g., basic types, mappings, sequences) or if one of them is a variable; otherwise, unification is impossible. If exactly one of them, say "e", is a variable, then the substitution list is augmented by the substitution of "f" for "e", and the common instance is "f." If the type expressions are of the same kind, then they have to be examined more closely. If they are both basic types, they can be unified if the types are identical; otherwise, unification fails. If they are both sequence or mapping types, then they unify if and only if the constituent types (i.e., the base types for sequences, the domain and range types for mappings) unify, so "unify" must be applied recursively in these cases. If both "e" and "f" are variables, then they can be unified; if they are different variables, then the substitution list must be augmented to indicate that one variable has been substituted for the other.

The Ernest code for the function "unify" is as follows:

```
define  unify( S, m, n ) =

--  Given substitution S, unify type expressions "m" and "n."
--  Assume type variables in "m" and "n" are distinct and
--  universally quantified.

    if e_kind = f_kind then
       if e_kind = basic then
          if e_elem = f_elem then <S,e>
          else <S,error>
       else if e_kind = seq then
          if u_elem = error then <S,error>
          else <S_seq,<seq,u_elem>>
       else if e_kind = mapping then
          if u_dom = error |
             u_rng = error then <S,error>
          else <S_rng,<mapping,rep(u_dom,S_rng),u_rng>>
       else if e_kind = var then
          if e_elem = f_elem then <S,e>
          else <insert(<e,f>,S),f>
       else <S,error>
    else if e_kind = var then <insert(<e,f>,S),f>
    else if f_kind = var then <insert(<f,e>,S),e>
    else <S,error>

where  e = rep(m,S)

and  e_kind:e_val = e

and  <e_elem> = e_val    -- for basic types and sequences

and  <e_dom,e_rng> = e_val    -- for mappings

and  f = rep(n,S)

and  f_kind:f_val = f

and  <f_elem> = f_val    -- for basic types and sequences

and  <f_dom,f_rng> = f_val    -- for mappings

and  <S_seq,u_elem> = unify(S,e_elem,f_elem)    -- for sequences

and  <S_dom,u_dom> = unify(S,e_dom,f_dom)    -- for mappings

and  <S_rng,u_rng> = unify(S_dom,e_rng,f_rng)
```

"Unify" calls two functions, "insert" and "rep." "Insert" returns the augmented substitution list obtained by adding the substitution item "x" to the list "S." "Rep" returns the representative for the type expression "e" in the substitution list "S." If "e" is a variable, "rep" searches through "S" for the type "e"; otherwise it simply returns "e", since only variables

are replaced in substitutions. The Ernest definitions for these two functions are as follows:

```
define  insert( x, S ) =

--  Insert equivalence pair "<v,r>" into substitution "S."

    if r_kind=var then x:S
    else insrt(x,S)

    where  <v,r_kind:r_val> = x

    and  insrt(x,S) = if S=<> then <x>
                      else if r_kind=var then a:insrt(x,t)
                      else x:S

        where  <v,r_kind:r_val> = a

        and  a:t = S


define  rep( e, S ) =

--  Find the representative of expression "e" in substitution
--  list "S."

    if e_kind=var then
       if s=<> then e
       else if e=v then rep(r,t)
       else rep(e,t)
    else e

    where  e_kind:e_val = e
```

The Function "Type_Check"

In the preceding section, we presented the Ernest code for unifying types. In this section, we give the Ernest code to do the actual type checking.

We assume that the Ernest code is in the form of an abstract syntax tree. Thus each expression has associated with it a "where" clause (which may be empty) and is tagged to indicate whether it is a literal, an identifier, or a function application. We further assume that the function applications have been Curried (i.e., that the applications are to one argument). This is a normalization, rather than a restriction, since any function application can be expressed as a sequence of Curried function applications. Thus, "2+3" is expressed as

```
<application, <application,"+",2>, 3>
```

The abstract syntax tree is built by pass 1 of the Ernest compiler.

There are two auxiliary structures that are necessary to support type checking. The substitution list, discussed above, contains information about which type variables are bound to which types. The environment contains the types of previously defined identifiers and functions, and it also contains the list of "fresh" variable names.

We can now describe how the type checking function works. The input expression "expr" is split into its component parts "expr_part" and "where_part", where "expr_part" is the actual expression part and "where_part" is the associated list of where clauses. The where clauses list is processed (as described below) to provide a new substitution list and environment "where_subst" and "where_env." The type checking function is split into three cases, depending on "expr":

- If "expr_part" is a literal value, then its type is the type of the literal.

- If "expr_part" is an identifier, then its type is determined by looking it up in the environment. The returned type is an instantiation of the type (that is, one in which the universally quantified variables within it are all replaced by fresh variables). This step is necessary because the variables within the type definition of a polymorphic function are, in effect, universally quantified variables.

- If "expr_part" is an application of function "func" to argument "arg", then its type is the type returned by "func" when applied to "arg." This value is determined as follows. A fresh type variable, "new_vbl", is created to represent the type of the return value. Both "func" and "arg" are type checked, yielding respective types "func_type" and "arg_type." The type expression "func_type" is unified with the type expression

  arg_type -> new_vbl

An example may help to illustrate this last case. Suppose that the application is "head(int_list)", where "int_list" is of type "seq(integer)." "Head" is of type

seq(alpha) -> alpha

If "beta" is the new type variable allocated, then we must unify the type of "head" with

seq(integer) -> beta

The result of this unification is the substitution of "integer" for "beta."

The Ernest code for "type_check" is as follows:

```
define  type_check(expr,subst,env) =

--  Given substitution list "subst" and environment "env",
--  determine the type of the expression "expr."  The value
--  returned is the list <T,substl,envl> where "T" is
--  the type of "expr", "substl" is the new substitution
--  list, and "envl" is the new environment.

    if expr_kind = literal then
       <literal_type(litval),where_subst,where_env>
    else if expr_kind = ident then
       <inst_type,subst,inst_env>
    else  -- expr_kind = application
       <new_vbl,func_subst,new_env>

    where  <func_subst, func_type> =
              unify(arg_subst,funcid_type,
                    <mapping,arg_type,new_vbl>)

    and  <new_vbl, new_env> = freshvbl(arg_env)

    and  <arg_type,arg_subst,arg_env> =
            type_check(<arg,<>>,funcid_subst,funcid_env)

    and  <funcid_type,funcid_subst,funcid_env> =
            type_check(<func,<>>,where_subst,where_env)

    and  <where_subst,where_env> =
            typecheck_wheres(where_part,subst,env)

    and  <expr_part,where_part> = expr

    and  expr_kind:expr_val = expr_part

    and  <litval> = expr_val    -- for literals

    and  <inst_type,inst_env> =
            remove_quant(id_type(idval, env), env)

    and  <idval> = expr_val    -- for identifiers

    and  <func,arg> = expr_val    -- for applications
```

The function "typecheck_wheres" checks the type of each definition in a where clause and returns the updated substitution list and environment. Its definition follows.

```
define  typecheck_wheres(wheres,subst,env) =
```

```
-- Given substitution list "subst" and environment "env",
-- augment them with the results of type checking the
-- given list of where clauses (definitions). The
-- returned value is the pair <substl,envl> where
-- "substl" is the new substitution list and "envl" is
-- the new environment.

    if wheres = <> then <subst,env>
    else typecheck_def(head(wheres),tail_subst,tail_env)

    where  <tail_subst,tail_env> =
              typecheck_wheres(tail(wheres),subst,env)
```

The function "typecheck_def" checks the type of a single definition and returns the updated substitution list and environment. It does this in the following fashion. The definition is in the form "<name,args,expr>", where "name" is the name of the identifier or function being defined, "args" is the list of arguments (which may be empty), and "expr" is the defining expression. A list of new type variables, "arg_vbls", is created, containing one type variable for each argument in "args." The expression "expr" is type checked, yielding a type "epxr_type." The type of the function being defined is then that type which, given arguments of the types constituting the list "arg_vbls", produces an argument of type "expr_type." This new type information is inserted in the updated environment that is returned.

As an example of how "typecheck_def" works, consider the function definition

```
define  f(x,y)  =  x ^ head(y)
```

("^" is the string concatenation operator.) The list "arg_vbls" consists of two fresh type variables for the arguments "x" and "y", say "alpha" and "beta." Type checking of the expression on the right hand side yields the type expression "string", and the substitutions "string" for "alpha" and "seq(string)" for "beta." Thus, "f" takes one argument of type "string" and another of type "seq(string)" and returns a result of type "string", or

```
string -> (seq(string) -> string)
```

The Ernest code for "typecheck_def" is as follows:

```
define  typecheck_def(def,subst,env) =

-- Type check a definition. The returned value is the
-- pair <substl,envl> where "substl" is the new
-- substitution list and "envl" is the new environment.

    <expr_subst,newenv>
```

```
        where  newenv = insert_env(name,funcdeftype,expr_env)

        and  funcdeftype = functype(arg_vbls,expr_type,
                                    expr_env,expr_subst)

        and  <expr_type,expr_subst,expr_env> =
                typecheck(expr,subst,arg_env)

        and  <arg_vbls,arg_env> = addargs(args,env)

        and  <name,args,expr> = def
```

The functions "func_type", which builds the type of a function from the types of its arguments and its return value, and "add_args", which assigns type variables to the arguments and updates the environment with the assignments, are defined as follows:

```
    define  func_type(arg_vbls,expr_type,env,subst) =

    --  Return the type of a function, given the types of its
    --  arguments ("arg_vbls") and the type of its returned
    --  value ("expr_type").

        if arg_vbls = <> then rep(expr_type,subst)
        else <mapping,rep(first_argvbl,subst),
              func_type(rest,expr_type,env,subst)>

        where  first_argvbl:rest = arg_vbls


    define  add_args(args,env) =

    --  Add type variables corresponding to the list of
    --  arguments ("args") to the environment.  The returned
    --  value is <vbls,env1>, where "vbls" is the list of type
    --  variables added and "env1" is the new environment.

        if args = <> then <<>,env>
        else <new_vbl:rest_vbls,ins_env>

        where  ins_env = insert_env(first_arg,new_vbl,new_env)

        and  <new_vbl,new_env> = freshvbl(rest_env)

        and  <rest_vbls,rest_env> = add_args(rest,env)

        and  first_arg:rest = args
```

The following low-level functions were used without being defined in the above definitions:

- "Literal_type(litval)" returns the type of the literal value "litval."

- "Id_type(idval,env)" returns the type of the identifier "idval" by looking it up in the environment "env."

- "Freshvbl(env)" creates a fresh variable, returning the newly created variable and the updated environment.

- "Remove_quant(t,env)" gives type expression "t" a unique set of fresh variables and returns the new type expression and an updated environment. For example, it would replace

  alpha -> (alpha -> integer)

  by

  gamma -> (gamma -> integer)

  where "gamma" is a fresh variable.

- "Insert_env(name,type,env)" returns a new environment which includes the information that the identifier named "name" has type "type."

These functions are straightforward to implement.

Two details have been glossed over in the above description. One is that the types of Ernest primitive functions must be defined before any other functions can be type checked. This can easily be accomplished by preloading the environment with these type definitions. The other is that we have not discussed the scope of identifiers and their visibility within an environment. We assume these details are taken care of by the functions that interact with the environment.

In addition, two type construction operations that are essential for a useful language, enumerated types and discriminated unions (variant records), have not been addressed. We believe these two type classes can be integrated into our type checking algorithm without great difficulty.

Correctness and Efficiency

There is much to be said for the simplicity and clarity of functions defined without type declarations and, indeed, this should continue to be an option. Nevertheless, there are two strong arguments for allowing users to provide type information: correctness and efficiency.

Correctness. If the user is permitted to specify the types of functions and constrain the allowable operations on data, this additional information can be used to help detect program errors. For example, it is probably not meaningful to add telephone

numbers to Social Security numbers, even if they are both represented by integer values. The type checker we have described cannot prevent such abuses.

Efficiency. There may be situations in which a user, knowing that the full generality of a polymorphic function is not necessary, is willing to trade flexibility for efficiency. For instance, if a function involves only arithmetic operations on its arguments, the type checker will determine that the arguments can be either real or integer. The decision to perform real or integer operations will be made at run-time, incurring some overhead. If the user could optionally specify that all arguments are reals, for example, the compiler could generate more efficient code that handles only real values. An attempt to call the function with integer arguments, however, would result in a type violation.

Improvements can be made in type checking thoroughness and in run-time efficiency by selectively introducing type declarations. The language changes required to support this capability are subjects for further research and development.

Conclusions

We have presented a program for type checking Ernest programs, which is written in Ernest. The program actually derives the type of functions, so that function definitions do not require type declarations. The type checker supports polymorphic functions and derives the most general interpretation of each function's type. In addition, the text of this program very closely follows textbook examples of type-derivation and type-checking algorithms.

References

1. Meeson, R., *Ernest User's Guide*, Computer Technology Associates, July 1986.

2. Curry, H.B., and R. Feys, *Combinatory Logic*, North-Holland, Amsterdam, 1958.

3. Milner, R., "A Theory of Type Polymorphism in Programming", *J. of Computer and System Sciences*, Vol. 17, 1978, pp. 348-375.

4. Aho, A., R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

Annex D. BIBLIOGRAPHY

Arvind, Kathail, V., and Pingali, K.K., "Sharing of Computation in Functional Language Implementations", Proceedings of the ACM International Workshop on High-Level Language Computer Architecture, May, 1984.

Augustsson, L., "A Compiler for Lazy ML", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 218-227.

Augustsson, L., "Compiling Pattern Matching", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 368-381.

Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", Communications of the ACM, vol. 21, no. 8, August, 1978, pp. 613-641.

Backus, J., "Function Level Programs as Mathematical Objects", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 1-10.

Backus, J., "Programming Language Semantics and Closed Applicative Languages", Conference Record of the ACM Symposium on Principles of Programming Languages, October, 1973, pp. 71-86.

Bellegarde, F., "Rewriting Systems on FP Expressions that Reduce the Number of Sequences They Yield", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 63-73.

Bellot, P., "High Order Programming in Extended FP", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 65-80.

Berkling, K.J., "Reduction Languages for Reduction Machines", Proceedings of the Second Annual IEEE Symposium on Computer Architecture, January, 1975, pp. 133-145.

Bohm, C., "Combinatory Foundation of Functional Programming", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 29-36.

Brownbridge, D.R., "Cyclic Reference Counting for Combinator Machines", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 273-288.

Buneman, P., Nikhil, R., and Frankel, R., "A Practical Functional Programming System for Databases", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 195-201.

Burge, W.H., "Combinatory Programming and Combinatorial Analysis", IBM Journal of Research and Development, vol. 16, no. 5, September, 1972, pp. 450-461.

Burge, W.H., Recursive Programming Techniques, Addison-Wesley Publishing Co., Reading, Mass., 1975.

Burge, W.H., "Stream Processing Functions", IBM Journal of Research and Development, vol. 19, no. 1, January, 1975, pp. 12-25.

Burstall, R.M., and Darlington, John, "A Transformation System for Developing Recursive Programs", Journal of ACM, vol.24, no. 1, January, 1977, pp. 44-67.

Burstall, R.M., MacQueen, D.B., and Sannella, D.T., "HOPE: An Experimental Applicative Language", Proceedings of the 1980 ACM LISP Conference, August, 1980, pp. 136-143.

Burton, F.W., and Sleep, M.R., "Executing Functional Programs on a Virtual Tree of Processors", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 187-194.

Bush, V.J., and Gurd, J.R., "Transforming Recursive Programs for Execution on Parallel Machines", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 350-367.

Cardelli, L., "Compiling a Functional Language", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 208-217.

Cartwright, R., and Donahue, J., "The Semantics of Lazy (and Industrious) Evaluation", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 253-264.

Clack, C., and Peyton Jones, S.L., "Strictness Analysis -- A Practical Approach", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 35-49.

Clarke, T.J.W., Gladstone, P.J.S., MacLean, C.D., and Norman, A.C., "SKIM -- The S, K, I, Reduction Machine", Proceedings of the 1980 ACM LISP Conference, August, 1980, pp. 128-135.

Coppo, M., "An Extended Polymorphic Type System for Applicative Languages", Mathematical Foundations of Computer Science, Proceedings 1980, Dembinski, P., Ed., LNCS, vol. 88, Springer-Verlag, 1980, pp. 194-204.

Curry, H.B., and Feys, R., Combinatory Logic, vol. 1, North-Holland Publishing Co., Amsterdam, 1958.

Cousineau, G., Curien, P.L., and Mauny, M., "The Categorical Abstract Machine", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 50-64.

Dannenberg, R.B., "Arctic: A Functional Language for Real-Time Control", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 96-103.

Damas, L., and Milner, R., "Principal Type-Schemes for Functional Programs", Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, January, 1982, pp. 207-212.

Darlington, J., Henderson, P., and Turner, D.A., Eds., Functional Programming and its Applications: An Advanced Course, Cambridge University Press, Cambridge, 1982.

Darlington, J., and Reeve, M., "ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages", Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 65-75.

Dennis, J.B., "Programming Generality, Parallelism and Computer Architecture", Information Processing 68: Proceedings of IFIP Congress 1968, Morrell, A.J.H., Ed., vol. 1, North-Holland Publishing Co., Amsterdam, pp. 484-492.

Dosch, W., and Moller, B., "Busy and Lazy FP with Infinite Objects", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 282-292.

Feldman, G., "Functional Specifications of a Text Editor", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 37-46.

Frank, G.A., "Specification of Data Structures for FP Programs", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 221-228.

Friedman, D.P., and Wise, D.S., "CONS Should Not Evaluate its Arguments", Automata, Languages and Programming, Michaelson and Milner, Eds., Edinburgh University Press, Edinburgh, 1976, pp. 257-284.

Friedman, D.P., and Wise, D.S., "An Approach to Fair Applicative Multiprogramming", Semantics of Concurrent Computation, G. Kahn, Ed., LNCS, vol. 70, Springer-Verlag, 1979, pp. 203-225.

Georgeff, M.P., "A Scheme for Implementing Functional Values on a Stack Machine", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 188-195.

Givler, J.S., and Kieburtz, R.B., "Schema Recognition for Program Transformations", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 74-84.

Grit, D.H., and Page, R.L., "Performance of a Multiprocessor for Applicative Programs", Proceedings of Performance 80: The Seventh IFIP Working Group International Symposium on Computer Performance Modelling, Measurement, and Evaluation, ACM SIGMETRICS, vol. 9, no. 2, May, 1980, pp. 181-189.

Gurd, J., and Watson, I., "Data Driven System for High Speed Parallel Computing - Part 1: Structuring Software for Parallel Execution", Computer Design, vol. 19, no. 6, June, 1980, pp. 91-100.

Guttag, J., Horning, J., and Williams, J., "FP with Data Abstraction and Strong Typing", Proceedings of the 1981 ACM Conference on Functional Programming and Computer Architecture, October, 1981, pp. 11-24.

Harrison, P., Khoshnevisan, H., "Efficient Compilation of Linear Recursive Functions into Object Level Loops", Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, June, 1986, pp. 207-218.

Henderson, P., "Functional Geometry", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 179-187.

Henderson, P., Functional Programming: Application and Implementation, Prentice-Hall International, Inc., London, 1980.

Henderson, P., "Functional Programming, Formal Specification, and Rapid Prototyping', IEEE Transactions on Software Engineering, vol. se-12, no.2, February, 1986, pp. 241-250.

Henderson, P., and Morris, J.H., "A Lazy Evaluator", Conference Record of the Third ACM Symposium on Principles of Programming Languages, January, 1976, pp. 95-103.

Hudak, P., and Goldberg, B., "Experiments in Diffused Combinator Reduction", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 167-176.

Hudak, P., and Keller, R.M., "Garbage Collection and Task Deletion in Distributed Applicative Processing Systems", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 168-178.

Hudak, P., and Kranz, D., "A Combinator-based Compiler for a Functional Language", Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, January, 1984, pp. 121-132.

Hughes, J., "A Distributed Garbage Collection Algorithm", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 256-272.

Hughes, J., "Lazy Memo-functions", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 129-146.

Hughes, R.J.M., "Super Combinators: A New Implementation Method for Applicative Languages", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 1-10.

Islam, N., Myers, T.J., and Broome, P., "A Simple Optimizer for FP-like Languages", Proceedings of the 1981 ACM Conference on Functional Programming and Computer Architecture, October, 1981, pp. 33-39.

Jones, A.K. et al, "Programming Issues Raised by a Multiprocessor", Proceedings of the IEEE, vol. 66, no. 2, February, 1978, pp. 229-237.

Jones, N.D., and Muchnick, S.S., "A Fixed-Program Machine for Combinator Expression Evaluation", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 11-20.

Kapur, D., Musser, D.R., and Stepanov, A.A., "Operators and Algebraic Structures", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 59-63.

Katayama, T., "Type Inference and Type Checking for Functional Programming Languages: A Reduced Computation Approach", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 263-272.

Keller, R.M., "Divide and CONCer Data Structuring in Applicative Multi-processing Systems", Proceedings of the 1980 ACM LISP Conference, August, 1980, pp. 196-202.

Keller, R.M., and Lindstrom, G., "Applications of Feedback in Functional Programming", Proceedings of the 1981 ACM Conference on Functional Programming and Computer Architecture, October, 1981, pp. 123-130.

Keller, R.M., and Sleep, M.R., "Applicative Caching: Programmer Control of Object Sharing and Lifetime in Distributed Implementations of Applicative Languages", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 131-140.

Kennaway, J.R., and Sleep, M.R., "Expressions as Processes", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 21-28.

Kennaway, J.R., and Sleep, M.R., "Parallel Implementation of Functional Languages", Proceedings of the 1982 International Conference on Parallel Processing, August, 1982, pp. 168-170.

Kieburtz, R.B., and Shultis, J., "Transformation of FP Program Schemes", Proceedings of the 1981 ACM Conference on Functional Programming and Computer Architecture, October, 1981, pp. 41-48.

Landin, P.J., "A Correspondence Between Algol 60 and Church's Lambda-Notation, Part II", Communications of the ACM, vol. 8, no. 3, March, 1965, pp. 158-165.

Landin, P.J., "The Mechanical Evaluation of Expressions", Computer Journal, vol. 6, no. 4, January, 1964, pp. 308-320.

Landin, P.J., "The Next 700 Programming Languages", Communications of the ACM, vol. 9, no. 3, March, 1966, pp. 157-164.

Lindstrom, G., "Static Evaluation of Functional Programs, Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, June, 1986, pp. 196-206.

MacQueen, D.B., "Modules for Standard ML", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 198-207.

MacQueen, D.B., and Sethi, R., "A Semantic Model of Types for Applicative Languages", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 243-252.

Mago, G.A., "A Cellular Computer Architecture for Functional Programming", IEEE COMPCON Spring 80, February, 1980, pp. 179-187.

Mago, G.A., "A Network of Microprocessors to Execute Reduction Languages, Part 1", International Journal of Computer and Information Sciences, vol. 8, no. 5, October, 1979, pp. 349-385.

Mago, G.A., "Copying Operands Versus Copying Results: A Solution to the Problem of Large Operands in FFP's", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 93-97.

Mago, G.A., "Data Sharing in an FFP Machine", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 201-207.

Malachi, Y., Manna, Z., and Waldinger, R., "TABLOG: The Deductive-Tableau Programming Language", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 323-330.

McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1", Communications of the ACM, vol. 3, no. 4, April, 1960, pp. 184-195.

Milne, R.D., and Strachey, C., A Theory of Programming Language Semantics, Chapman and Hall, London, 1976.

Milner, R., "A Proposal for Standard ML", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 184-197.

Nikhil, R.S., "Practical Polymorphism", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 319-333.

Nordstrom, B., "Programming in Constructive Set Theory: Some Examples", Proceedings of the 1981 ACM Conference on Functional Programming and Computer Architecture, October, 1981, pp. 141-153.

Page, R.L., Conant, M.G., and Grit, D.H., "If-then-else as a Concurrency Inhibitor in Eager Beaver Evaluation", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 179-186.

Pargas, R.P., and Presnell, H.A., "Communication Along Shortest Paths in a Tree Machine", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 107-114.

Peterson, J.C., and Murray, W.D., "Parallel Computer Architecture Employing Functional Programming Systems", Proceedings of the ACM International Workshop on High-Level Language Computer Architecture, May, 1980, pp. 190-195.

Peyton Jones, S.L., "An Investigation of the Relative Efficiencies of Combinators and Lambda-expressions", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 150-158.

Pingali, K., and Arvind, "Efficient Demand-Driven Evaluation Part 1", ACM Transactions on Programming Languages and Systems vol.7, no.2, April, 1985, pp. 331-333.

Rosser, J.B., "Highlights of the History of the Lambda-Calculus", Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, August, 1982, pp. 216-225.

Schwartz, J.T., "Automatic Data Structure Choice in a Language of Very High Level", Conference Record of the Second ACM Symposium on Principles of Programming Languages, January, 1975, pp. 36-40.

Scott, D.S., "Logic and Programming Languages", Communications of the ACM, vol. 20, no. 9, September, 1977, pp. 634-641.

Scott, D., and Strachey, C., "Towards a Mathematical Semantics for Computer Languages", Proceedings of the Symposium on Computers and Automata, Microwave Research Institute Symposia Series, vol. 21, Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y., 1971, pp. 19-46.

Sheeran, M., "muFP: A Language for VLSI Design", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 104-112.

Sintzoff, M., "Proof-oriented and Applicative Valuations in Definitions of Algorithms", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 155-162.

Sleep, M.R., "Applicative Languages, Dataflow, and Pure Combinatory Code", IEEE COMPCON Spring 80, February, 1980, pp. 112-115.

Stanat, D.F., and Williams, E.H., "Optimal Associative Searching on a Cellular Computer", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 99-106.

Stoy, J.E., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, Cambridge, Mass, 1977.

Stoye, W.R., Clarke, T.J.W., and Norman, A.C., "Some Practical Methods for Rapid Combinator Reduction", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 159-166.

Tesler, L.G., and Enea, H.J., "A Language Design for Concurrent Processes", AFIPS Conference Proceedings, vol. 32, 1968, pp. 403-408.

Toole, D.M., "Implanting FFP Trees in Binary Trees", Proceedings of the 1981 ACM Conference on Functional Programming and Computer Architecture, October, 1981, pp. 115-122.

Treleaven, P.C., Brownbridge, D.R., and Hopkins, R.P., "Data-driven and Demand-driven Computer Architecture", ACM Computer Surveys, vol. 14, no. 1, March, 1982, pp. 93-143.

Treleaven, P.C., and Hopkins, R.P., "Decentralised Computation", Proceedings of the Eighth International Symposium on Computer Architecture, May, 1981, pp. 279-290.

Turchin, V.F., "The Concept of a Supercompiler", ACM Transactions on Programming Languages and Systems, vol. 8, no. 3, July, 1986, pp. 292-325.

Turner, D.A., "A New Implementation Technique for Applicative Languages", Software -- Practice and Experience, vol. 9, September, 1979, pp. 31-49.

Turner, D.A., "A Non-strict Functional Language with Polymorphic Types", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 1-16.

Turner, D.A., "The Semantic Elegance of Applicative Languages", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 85-92.

Ullman, J.D., "Some Thoughts about Supercomputer Organization", IEEE COMPCON Spring 84, February, 1984, pp. 424-432.

Vegdahl, S.R., "A Survey of Proposed Architectures for the Execution of Functional Languages", IEEE Transactions on Computers, vol. c-33, no. 12, December, 1984, pp. 1050-1071.

Wadler, P., "Applicative Style Programming, Program Transformation, and List Operators", Proceedings of the 1981 ACM Conference on Functional Programming Languages and Computer Architecture, October, 1981, pp. 25-32.

Wadler, P., "How to Replace Failure by a List of Successes", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 113-128.

Wadler, P., "Listlessness is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile Time", Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, August, 1984, pp. 45-52.

Young, M.F., "A Functional Language and Modular Architecture for Scientific Computing", Functional Programming Languages and Computer Architecture, Proceedings 1985, Jouannaud, J., Ed., LNCS, vol. 201, Springer-Verlag, 1985, pp. 305-318.

END

11-86

DTIC